目录

TypeScript Handbook（中文版）

TS TypeScript 2.9 RC (May 16, 2018)

> 从前打心眼儿里讨厌编译成JavaScript的这类语言，像Coffee，Dart等。 但是在15年春节前后却爱上了TypeScript。 同时非常喜欢的框架Dojo，Angularjs也宣布使用TypeScript做新版本的开发。 那么TypeScript究竟为何物？又有什么魅力呢？

TypeScript是Microsoft公司注册商标。

TypeScript具有类型系统，且是JavaScript的超集。 它可以编译成普通的JavaScript代码。 TypeScript支持任意浏览器，任意环境，任意系统并且是开源的。

TypeScript目前还在积极的开发完善之中，不断地会有新的特性加入进来。 因此本手册也会紧随官方的每个commit，不断地更新新的章节以及修改措词不妥之处。

如果你对TypeScript一见钟情，可以订阅~~and star~~本手册，及时了解ECMAScript 2015以及2016里新的原生特性，并借助TypeScript提前掌握使用它们的方式！ 如果你对TypeScript的爱愈发浓烈，可以与楼主一起边翻译边学习，*PRs Welcome!!!* 在相关链接的末尾可以找到本手册的Github地址。

目录

主要修改 (Latest 5 updates)

- 2017-11-07 新增章节：JavaScript文件里的类型检查
- 2017-05-16 新增章节：教程-5分钟了解TypeScript
- 2017-05-01 新增章节：教程-React
- 2016-11-27 新增章节：使用 `/// <reference types="..." />`
- 2016-11-23 新增章节：变量声明 - 展开操作符
- 2016-10-23 新增章节：高级类型 - 可辨识联合（Discriminated Unions）

相关链接

- TypeScript官网
- TypeScript on Github
- TypeScript语言规范
- 本手册中文版Github地址

Table of Contents

让我们使用TypeScript来创建一个简单的Web应用。

安装TypeScript

有两种主要的方式来获取TypeScript工具：

- 通过npm（Node.js包管理器）
- 安装Visual Studio的TypeScript插件

Visual Studio 2017和Visual Studio 2015 Update 3默认包含了TypeScript。 如果你的Visual Studio还没有安装TypeScript，你可以下载它。

针对使用npm的用户：

```
> npm install -g typescript
```

构建你的第一个TypeScript文件

在编辑器，将下面的代码输入到 `greeter.ts` 文件里：

```
function greeter(person) {
    return "Hello, " + person;
}

let user = "Jane User";

document.body.innerHTML = greeter(user);
```

编译代码

我们使用了 `.ts` 扩展名，但是这段代码仅仅是JavaScript而已。 你可以直接从现有的JavaScript应用里复制/粘贴这段代码。

在命令行上，运行TypeScript编译器：

```
tsc greeter.ts
```

输出结果为一个 `greeter.js` 文件，它包含了和输入文件中相同的JavsScript代码。 一切准备就绪，我们可以运行这个使用TypeScript写的JavaScript应用了！

接下来让我们看看TypeScript工具带来的高级功能。 给 `person` 函数的参数添加 `: string` 类型注解，如下：

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = "Jane User";

document.body.innerHTML = greeter(user);
```

类型注解

TypeScript里的类型注解是一种轻量级的为函数或变量添加约束的方式。 在这个例子里，我们希望 `greeter` 函数接收一个字符串参数。 然后尝试把 `greeter` 的调用改成传入一个数组：

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = [0, 1, 2];

document.body.innerHTML = greeter(user);
```

重新编译，你会看到产生了一个错误。

```
error TS2345: Argument of type 'number[]' is not assignable to p
arameter of type 'string'.
```

类似地，尝试删除 `greeter` 调用的所有参数。 TypeScript会告诉你使用了非期望个数的参数调用了这个函数。 在这两种情况中，TypeScript提供了静态的代码分析，它可以分析代码结构和提供的类型注解。

要注意的是尽管有错误，`greeter.js`文件还是被创建了。 就算你的代码里有错误，你仍然可以使用TypeScript。但在这种情况下，TypeScript会警告你代码可能不会按预期执行。

接口

让我们开发这个示例应用。这里我们使用接口来描述一个拥有`firstName`和`lastName`字段的对象。 在TypeScript里，只在两个类型内部的结构兼容那么这两个类型就是兼容的。 这就允许我们在实现接口时候只要保证包含了接口要求的结构就可以，而不必明确地使用`implements`语句。

```
interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person: Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = { firstName: "Jane", lastName: "User" };

document.body.innerHTML = greeter(user);
```

类

最后，让我们使用类来改写这个例子。 TypeScript支持JavaScript的新特性，比如支持基于类的面向对象编程。

让我们创建一个`Student`类，它带有一个构造函数和一些公共字段。 注意类和接口可以一起共作，程序员可以自行决定抽象的级别。

还要注意的是，在构造函数的参数上使用`public`等同于创建了同名的成员变量。

```
class Student {
    fullName: string;
    constructor(public firstName: string, public middleInitial: string, public lastName: string) {
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}

interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);
```

重新运行 `tsc greeter.ts` ，你会看到生成的JavaScript代码和原先的一样。TypeScript里的类只是JavaScript里常用的基于原型面向对象编程的简写。

运行TypeScript Web应用

在 `greeter.html` 里输入如下内容：

```
<!DOCTYPE html>
<html>
    <head><title>TypeScript Greeter</title></head>
    <body>
        <script src="greeter.js"></script>
    </body>
</html>
```

在浏览器里打开`greeter.html`运行这个应用！

可选地：在Visual Studio里打开`greeter.ts`或者把代码复制到TypeScript playground。 将鼠标悬停在标识符上查看它们的类型。 注意在某些情况下它们的类型可以被自动地推断出来。 重新输入一下最后一行代码，看一下自动补全列表和参数列表，它们会根据DOM元素类型而变化。 将光标放在`greeter`函数上，点击F12可以跟踪到它的定义。 还有一点，你可以右键点击标识，使用重构功能来重命名。

这些类型信息以及工具可以很好的和JavaScript一起工作。 更多的TypeScript功能演示，请查看本网站的示例部分。

ASP.NET Core

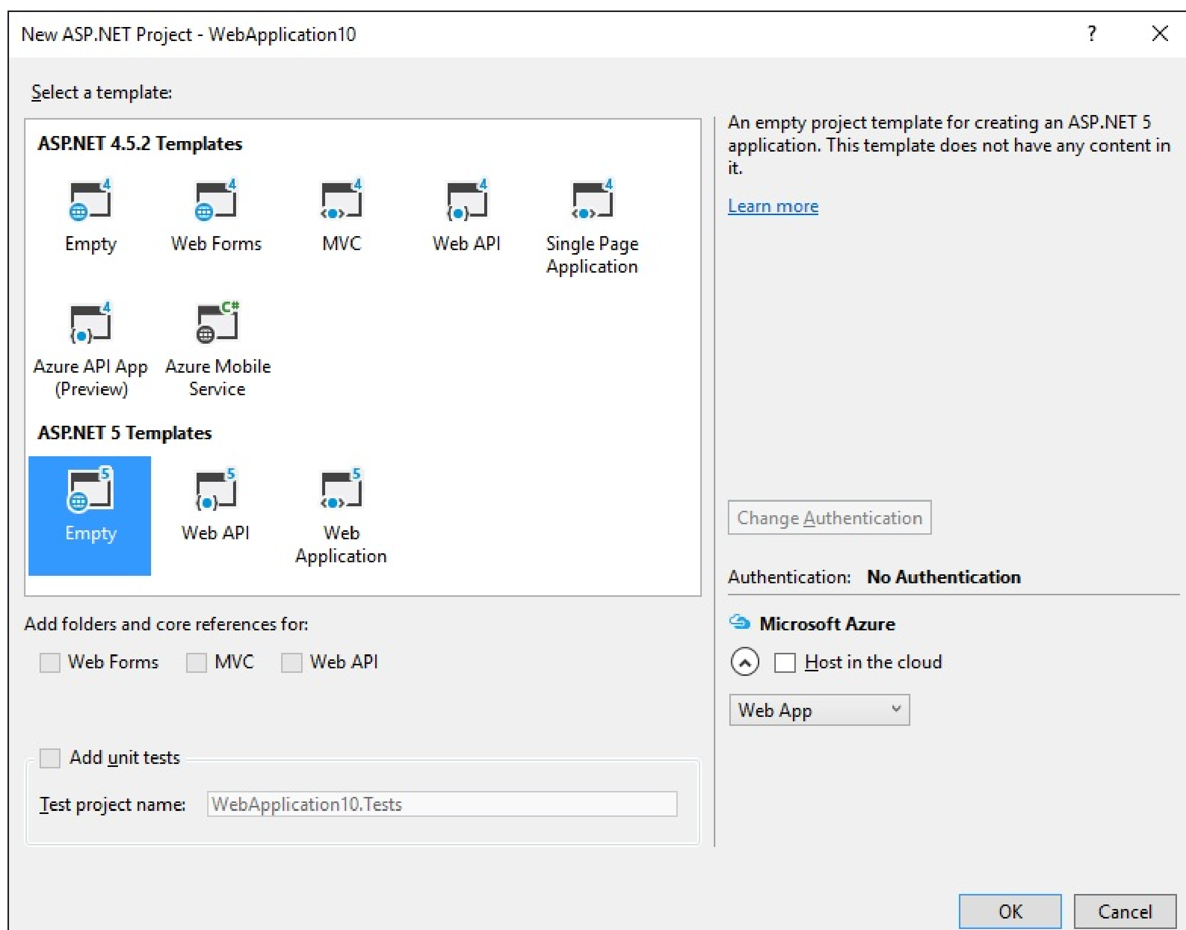
安装 ASP.NET Core 和 TypeScript

首先，若有需要请安装 ASP.NET Core。此篇指南需要使用Visual Studio 2015或2017。

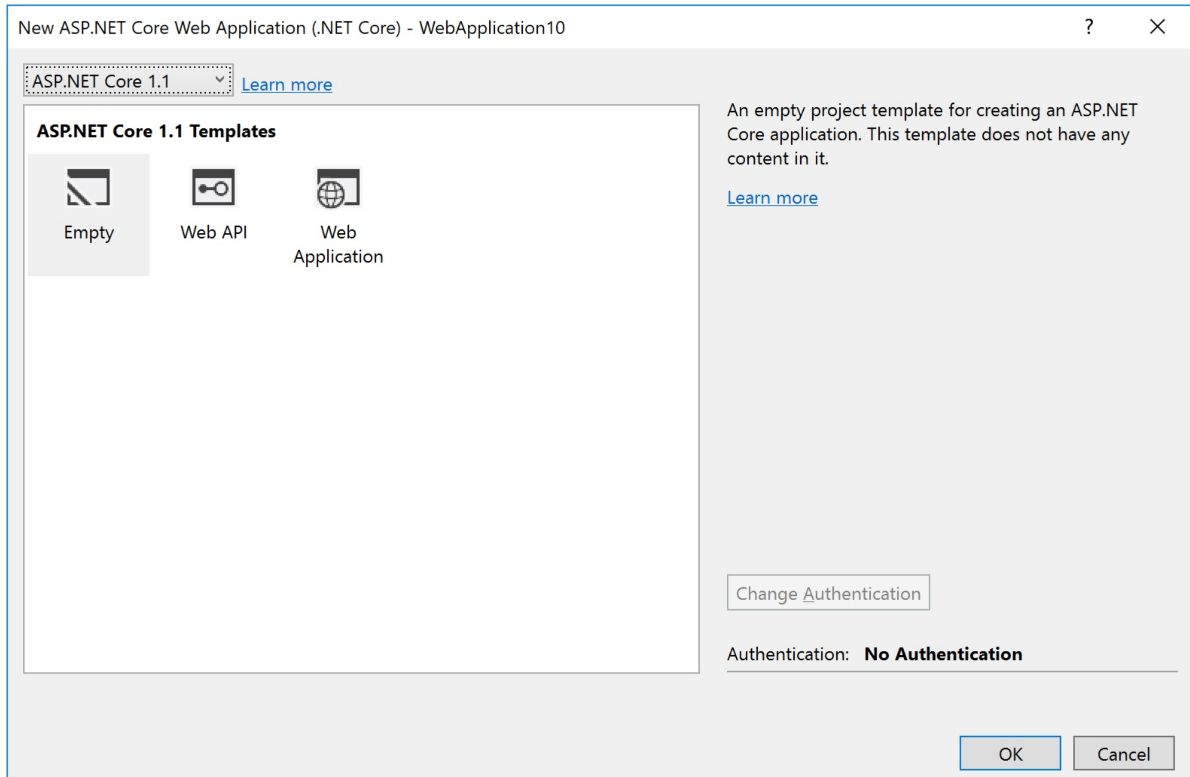
其次，如果你的Visual Studio不带有最新版本的TypeScript，你可以从这里安装。

新建工程

1. 选择 **File**
2. 选择 **New Project** （Ctrl + Shift + N）
3. 选择 **Visual C#**
4. 若使用VS2015，选择 **ASP.NET Web Application** > **ASP.NET 5 Empty**，并且取消勾选“Host in the cloud”，因为我们要在本地运行。



5. 若使用VS2017，选择 **ASP.NET Core Web Application (.NET Core)** > **ASP.NET Core 1.1 Empty**。



运行此应用以确保它能正常工作。

设置服务项

VS2015

在 `project.json` 文件的 `"dependencies"` 字段里添加:

```
"Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
```

最终的 dependencies 部分应该类似于下面这样：

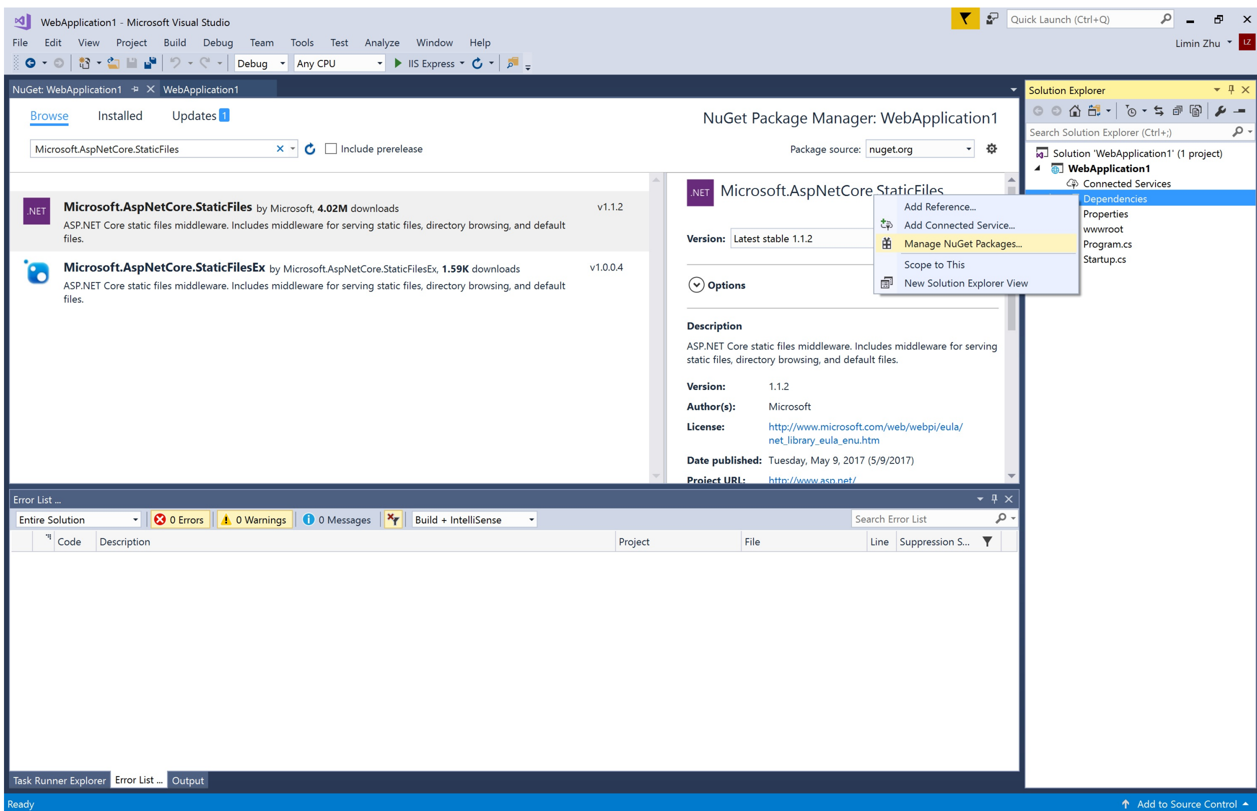
```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
  "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
},
```

用以下内容替换 `Startup.cs` 文件里的 `Configure` 函数：

```
public void Configure(IApplicationBuilder app)
{
    app.UseIISPlatformHandler();
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

VS2017

打开 **Dependencies** > **Manage NuGet Packages** > **Browse**。搜索并安装 `Microsoft.AspNetCore.StaticFiles` 1.1.2：



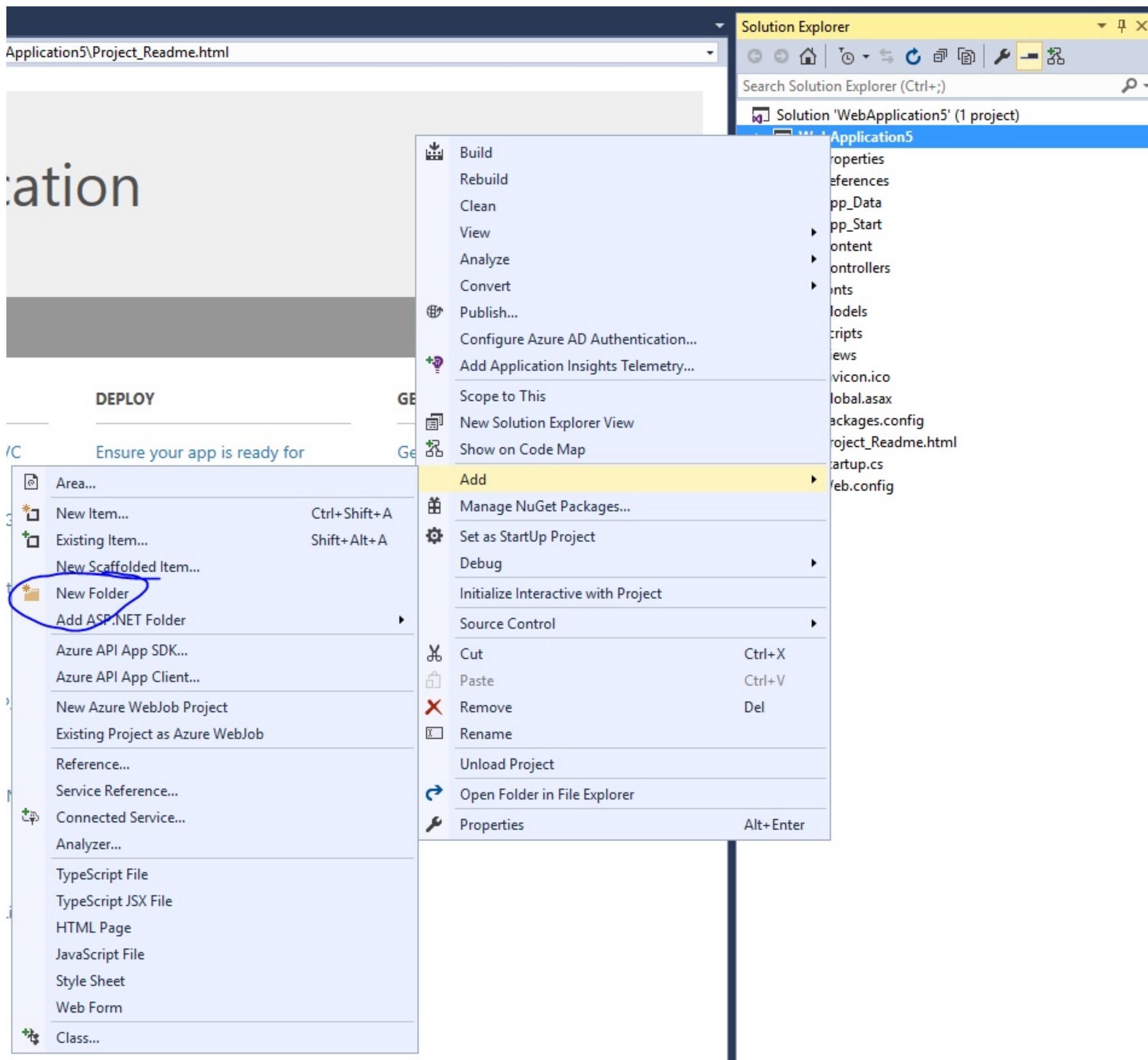
如下替换掉 `Startup.cs` 里 `Configure` 的内容：

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

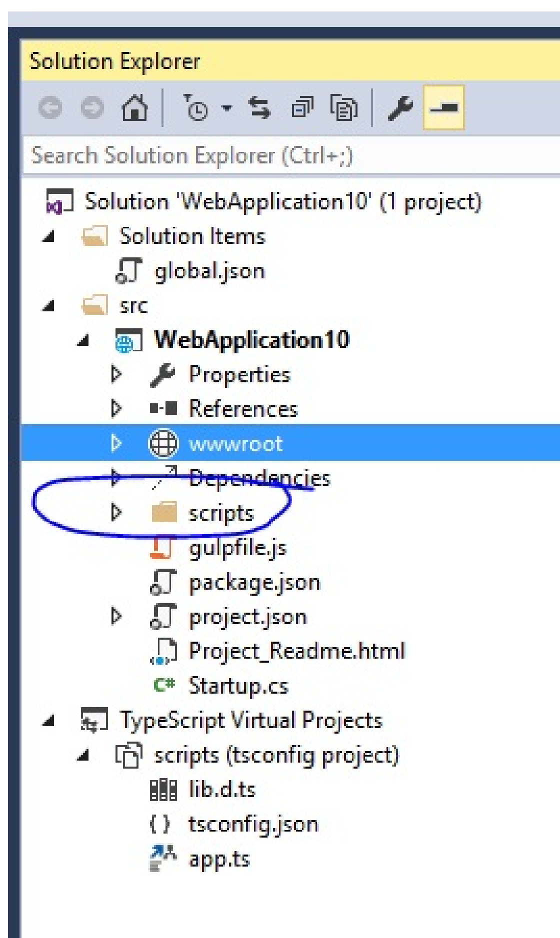
你可能需要重启VS，这样 `UseDefaultFiles` 和 `UseStaticFiles` 下面的波浪线才会消失。

添加 TypeScript

下一步我们为 TypeScript 添加一个文件夹。

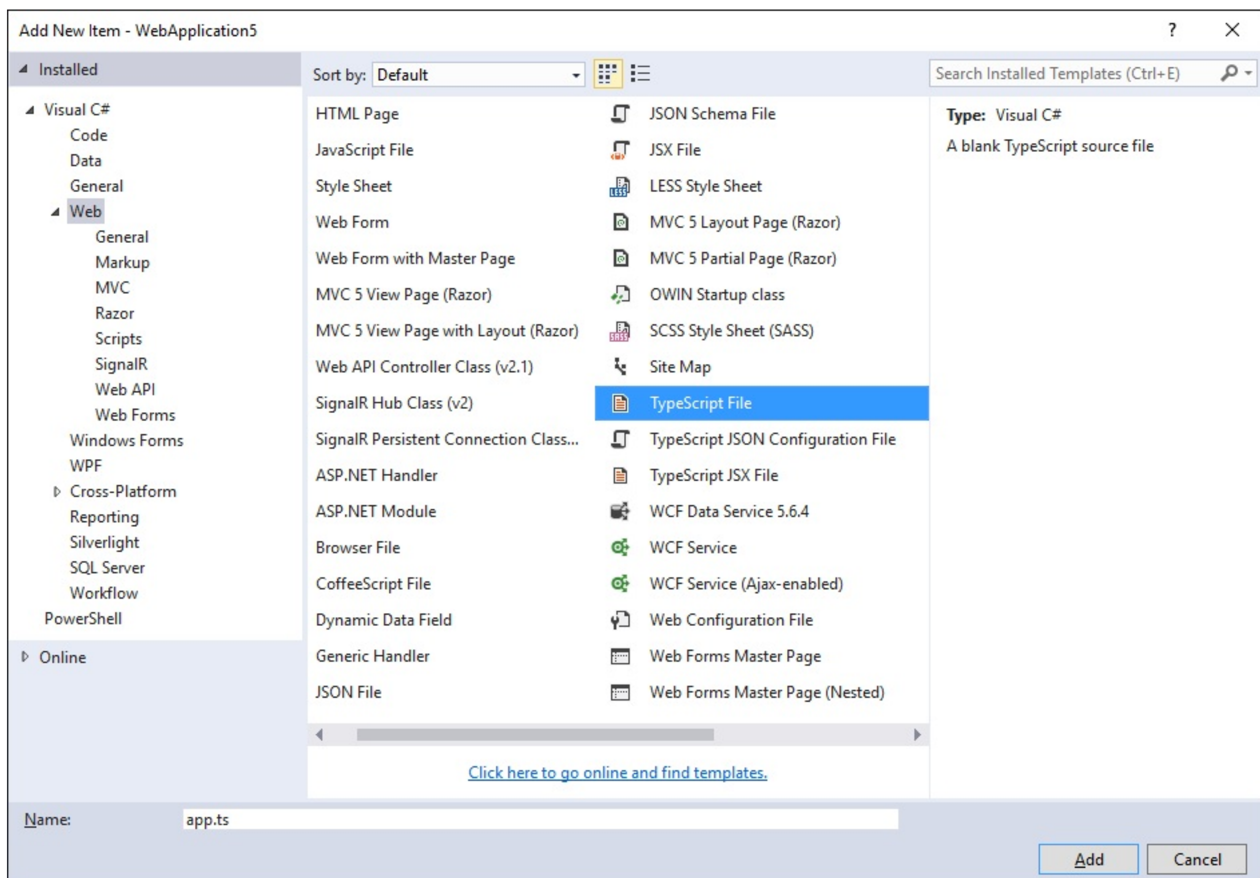


将文件夹命名为 `scripts` 。



添加 TypeScript 代码

在 `scripts` 上右击并选择**New Item**。接着选择**TypeScript File**（也可能 .NET Core 部分），并将此文件命名为 `app.ts` 。



添加示例代码

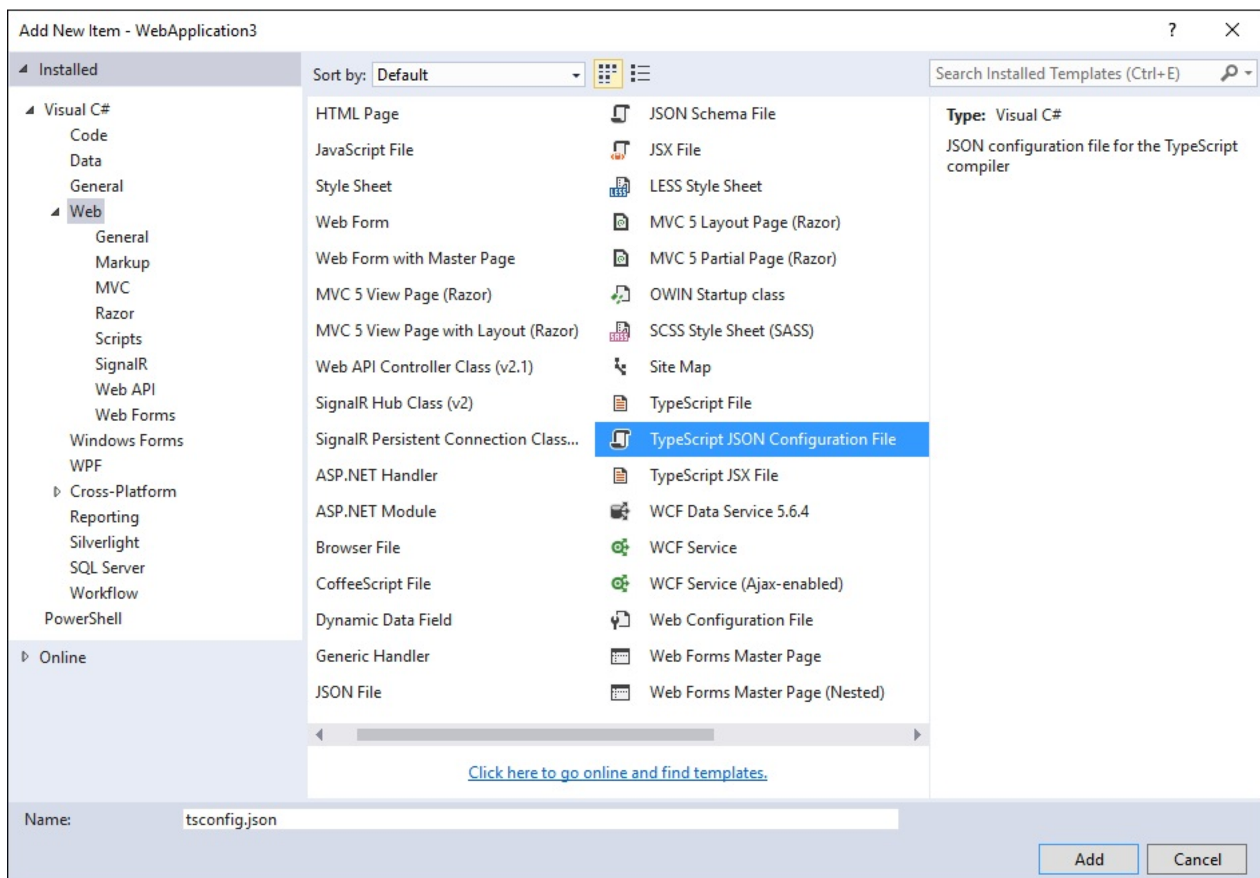
将以下代码写入app.ts文件。

```
function sayHello() {
  const compiler = (document.getElementById("compiler") as HTMLInputElement).value;
  const framework = (document.getElementById("framework") as HTMLInputElement).value;
  return `Hello from ${compiler} and ${framework}!`;
}
```

构建设置

配置 TypeScript 编译器

我们先来告诉TypeScript怎样构建。 右击scripts文件夹并选择**New Item**。 接着选择**TypeScript Configuration File**，保持文件的默认名字为 `tsconfig.json` 。



将默认的 `tsconfig.json` 内容改为如下所示：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5"
  },
  "files": [
    "./app.ts"
  ],
  "compileOnSave": true
}
```

看起来和默认的设置差不多，但注意以下不同之处：

1. 设置 `"noImplicitAny": true` 。
2. 显式列出了 `"files"` 而不是依据 `"excludes"` 。
3. 设置 `"compileOnSave": true` 。

当你写新代码时，设置 `"noImplicitAny"` 选项是个不错的选择 — 这可以确保你不会错写任何新的类型。 设置 `"compileOnSave"` 选项可以确保你在运行web程序前自动编译保存变更后的代码。

配置 NPM

现在，我们来配置NPM以使用我们能够下载JavaScript包。 在工程上右击并选择**New Item**。 接着选择**NPM Configuration File**，保持文件的默认名字为 `package.json` 。 在 `"devDependencies"` 部分添加"gulp"和"del"：

```
"devDependencies": {
  "gulp": "3.9.0",
  "del": "2.2.0"
}
```

保存这个文件后，Visual Studio将开始安装gulp和del。 若没有自动开始，请右击package.json文件选择**Restore Packages**。

设置 gulp

最后，添加一个新JavaScript文件 `gulpfile.js` 。 键入以下内容：

```
/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and us
ing Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId
=518007
*/

var gulp = require('gulp');
var del = require('del');

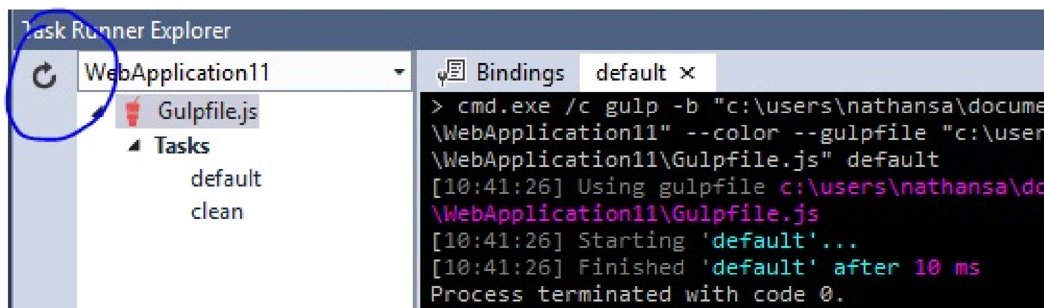
var paths = {
  scripts: ['scripts/**/*.js', 'scripts/**/*.ts', 'scripts/**/*.
map'],
};

gulp.task('clean', function () {
  return del(['wwwroot/scripts/**/*']);
});

gulp.task('default', function () {
  gulp.src(paths.scripts).pipe(gulp.dest('wwwroot/scripts'))
});
```

第一行是告诉Visual Studio构建完成后，立即运行'default'任务。 当你应答 Visual Studio 清除构建内容后，它也将运行'clean'任务。

现在，右击 `gulpfile.js` 并选择**Task Runner Explorer**。 若'default'和'clean'任务没有显示输出内容的话，请刷新explorer：



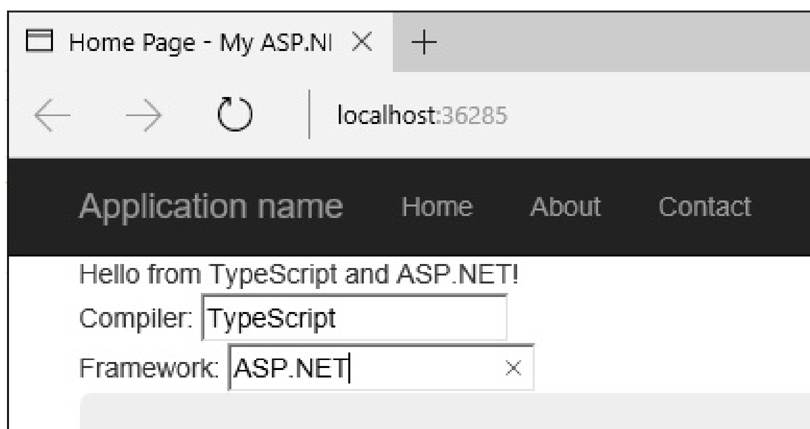
编写HTML页

在 `wwwroot` 中添加一个新建项 `index.html` 。在 `index.html` 中写入以下代码：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <script src="scripts/app.js"></script>
    <title></title>
</head>
<body>
    <div id="message"></div>
    <div>
        Compiler: <input id="compiler" value="TypeScript" onkeyup="document.getElementById('message').innerText = sayHello()" /><br />
        Framework: <input id="framework" value="ASP.NET" onkeyup="document.getElementById('message').innerText = sayHello()" />
    </div>
</body>
</html>
```

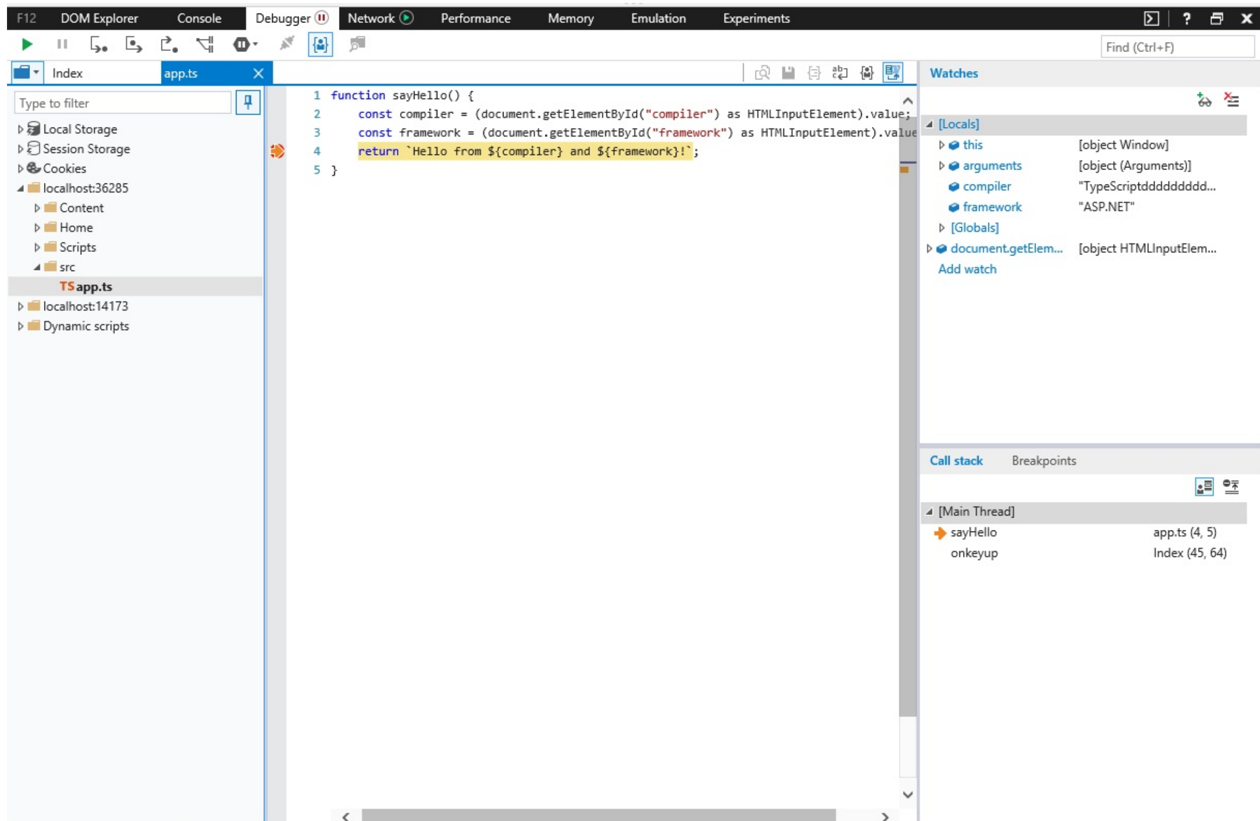
测试

1. 运行项目。
2. 在输入框中键入时，您应该看到一个消息：



调试

1. 在 Edge 浏览器中，按 F12 键并选择 **Debugger** 标签页。
2. 展开 localhost 列表，选择 scripts/app.ts
3. 在 `return` 那一行上打一个断点。
4. 在输入框中键入一些内容，确认TypeScript代码命中断点，观察它是否能正确地工作。



这就是你需要知道的在ASP.NET中使用TypeScript的基本知识了。 接下来，我们引入Angular，写一个简单的Angular程序示例。

添加 Angular 2

使用 NPM 下载依赖的包

添加Angular 2和SystemJS到 `package.json` 的 `dependencies` 里。

对于VS2015，新的 `dependencies` 列表如下：

```
"dependencies": {
  "angular2": "2.0.0-beta.11",
  "systemjs": "0.19.24",
  "gulp": "3.9.0",
  "del": "2.2.0"
},
```

若使用VS2017，因为NPM3反对同行的依赖（peer dependencies），我们需要把Angular 2同行的依赖也直接列为依赖项：

```
"dependencies": {
  "angular2": "2.0.0-beta.11",
  "reflect-metadata": "0.1.2",
  "rxjs": "5.0.0-beta.2",
  "zone.js": "^0.6.4",
  "systemjs": "0.19.24",
  "gulp": "3.9.0",
  "del": "2.2.0"
},
```

更新 tsconfig.json

现在安装好了Angular 2及其依赖项，我们需要启用TypeScript中实验性的装饰器支持。 我们还需要添加ES2015的声明，因为Angular使用core-js来支持像 `Promise` 的功能。 在未来，装饰器会成为默认设置，那时也就不再需要这些设置了。

添加 `"experimentalDecorators": true, "emitDecoratorMetadata": true` 到 `"compilerOptions"` 部分。 然后，再添加 `"lib": ["es2015", "es5", "dom"]` 到 `"compilerOptions"` ，以引入ES2015的声明。 最后，我们需要添加 `"./model.ts"` 到 `"files"` 里，我们接下来会创建它。 现在 `tsconfig.json` 看起来如下：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "target": "es5",
    "lib": [
      "es2015", "es5", "dom"
    ]
  },
  "files": [
    "./app.ts",
    "./model.ts",
    "./main.ts",
  ],
  "compileOnSave": true
}
```

将 Angular 添加到 gulp 构建中

最后，我们需要确保 Angular 文件作为 build 的一部分复制进来。 我们需要添加：

1. 库文件目录。
2. 添加一个 `lib` 任务来输送文件到 `wwwroot` 。
3. 在 `default` 任务上添加 `lib` 任务依赖。

更新后的 `gulpfile.js` 像如下所示：

```
/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and us
ing Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId
=518007
*/

var gulp = require('gulp');
var del = require('del');

var paths = {
    scripts: ['scripts/**/*.js', 'scripts/**/*.ts', 'scripts/**/
*.map'],
    libs: ['node_modules/angular2/bundles/angular2.js',
           'node_modules/angular2/bundles/angular2-polyfills.js'
,
           'node_modules/systemjs/dist/system.src.js',
           'node_modules/rxjs/bundles/Rx.js']
};

gulp.task('lib', function () {
    gulp.src(paths.libs).pipe(gulp.dest('wwwroot/scripts/lib'));
});

gulp.task('clean', function () {
    return del(['wwwroot/scripts/**/*']);
});

gulp.task('default', ['lib'], function () {
    gulp.src(paths.scripts).pipe(gulp.dest('wwwroot/scripts'));
});
```

此外，保存了此gulpfile后，要确保 Task Runner Explorer 能看到 `lib` 任务。

用 TypeScript 写一个简单的 Angular 应用

首先，将 `app.ts` 改成：

```
import {Component} from "angular2/core"
import {MyModel} from "./model"

@Component({
    selector: `my-app`,
    template: `<div>Hello from {{getCompiler()}}</div>`
})
export class MyApp {
    model = new MyModel();
    getCompiler() {
        return this.model.compiler;
    }
}
```

接着在 `scripts` 中添加TypeScript文件 `model.ts` :

```
export class MyModel {
    compiler = "TypeScript";
}
```

再在 `scripts` 中添加 `main.ts` :

```
import {bootstrap} from "angular2/platform/browser";
import {MyApp} from "./app";
bootstrap(MyApp);
```

最后，将 `index.html` 改成：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <script src="scripts/lib/angular2-polyfills.js"></script>
    <script src="scripts/lib/system.src.js"></script>
    <script src="scripts/lib/rx.js"></script>
    <script src="scripts/lib/angular2.js"></script>
    <script>
    System.config({
        packages: {
            'scripts': {
                format: 'cjs',
                defaultExtension: 'js'
            }
        }
    });
    System.import('scripts/main').then(null, console.error.bind(
console));
    </script>
    <title></title>
</head>
<body>
    <my-app>Loading...</my-app>
</body>
</html>
```

这里加载了此应用。 运行 ASP.NET 应用，你应该能看到一个div显示"Loading..."紧接着更新成显示"Hello from TypeScript"。

ASP.NET 4

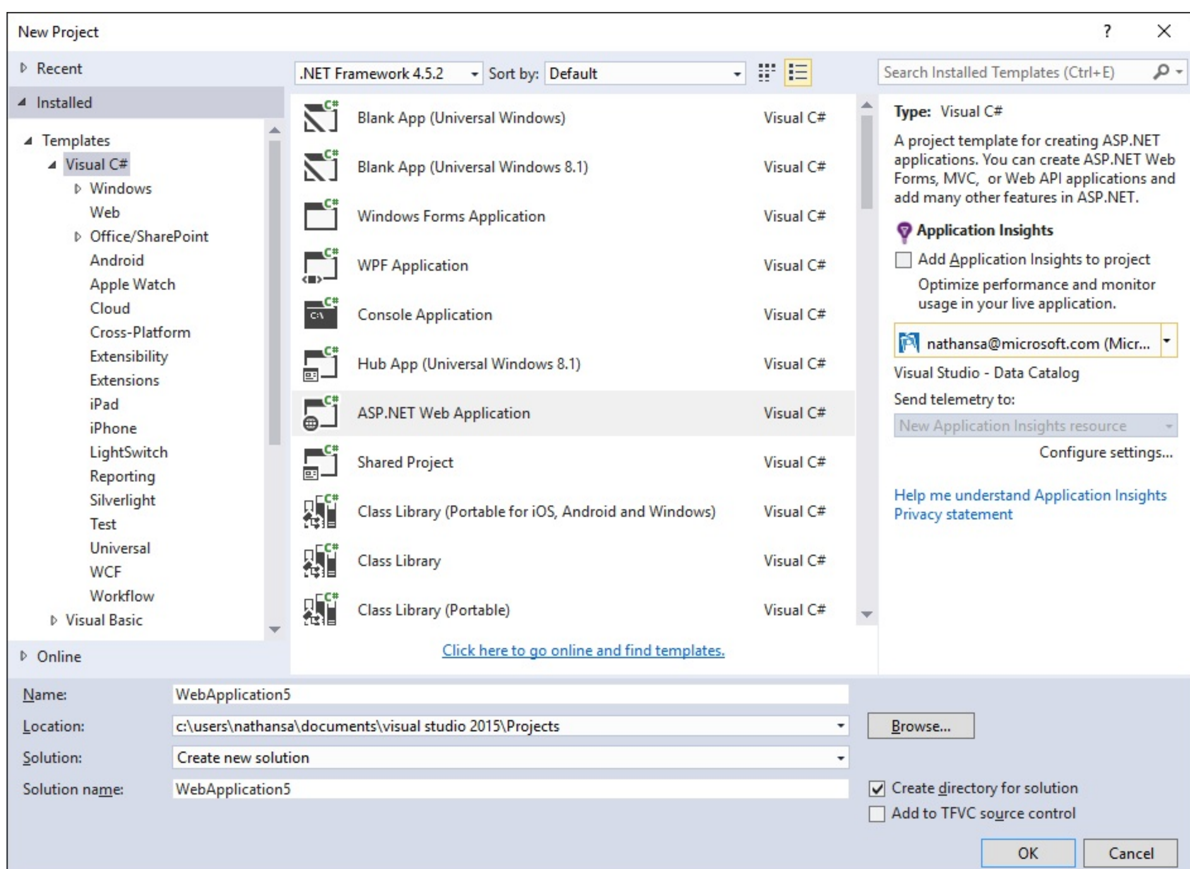
> 注意： 此教程已从官方删除

安装 TypeScript

如果你使用的 Visual Studio 版本还不支持 TypeScript， 你可以安装 Visual Studio 2015 或者 Visual Studio 2013。 这个快速上手指南使用的是 Visual Studio 2015。

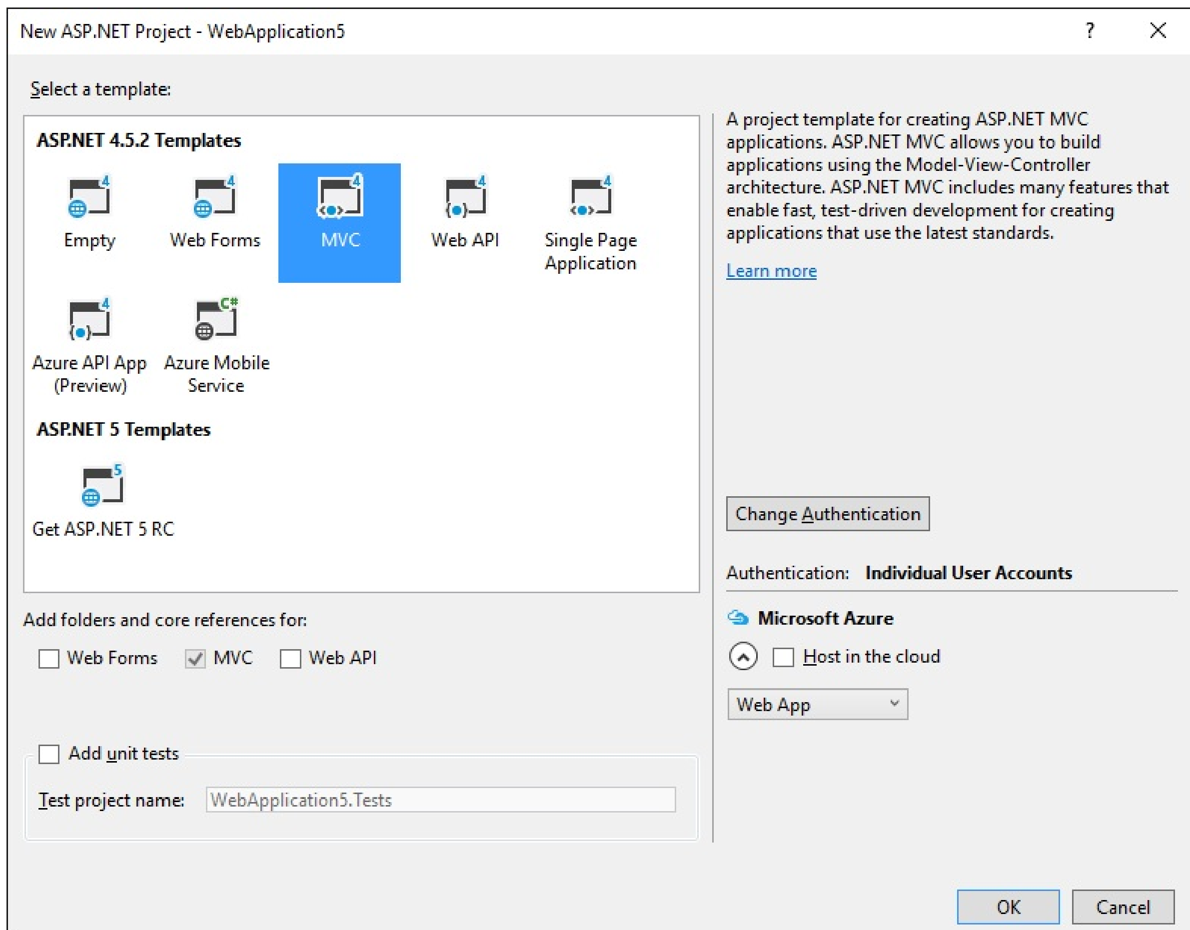
新建项目

1. 选择 **File**
2. 选择 **New Project**
3. 选择 **Visual C#**
4. 选择 **ASP.NET Web Application**



5. 选择 **MVC**

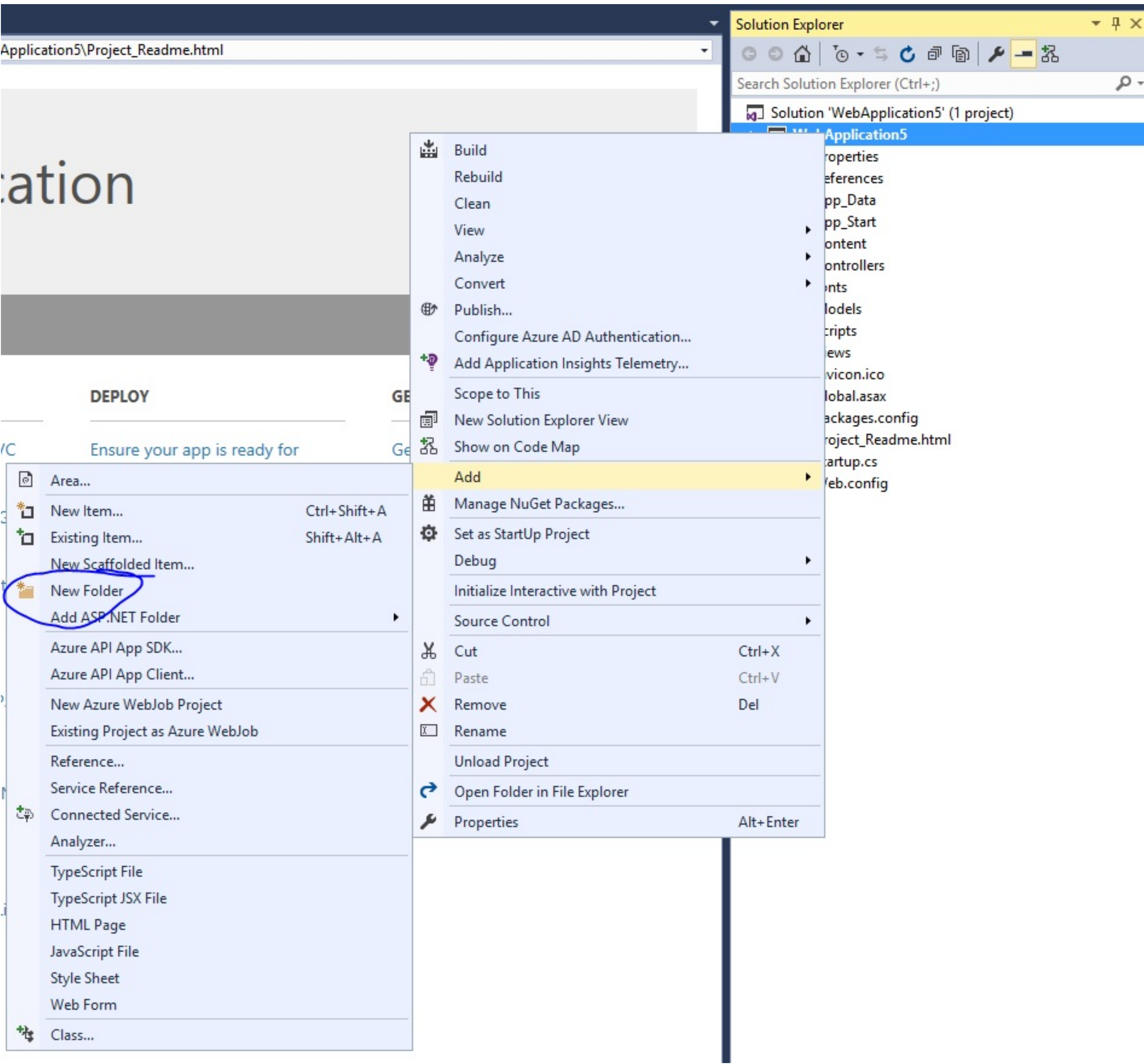
取消复选 "Host in the cloud" 本指南将使用一个本地示例。



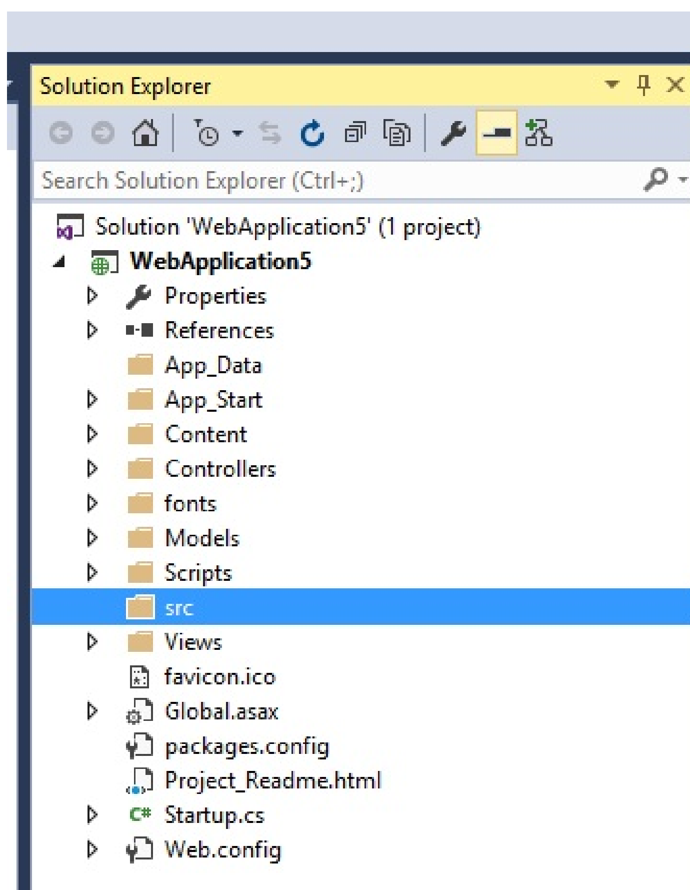
运行此应用以确保它能正常工作。

添加 TypeScript

下一步我们为 TypeScript 添加一个文件夹。

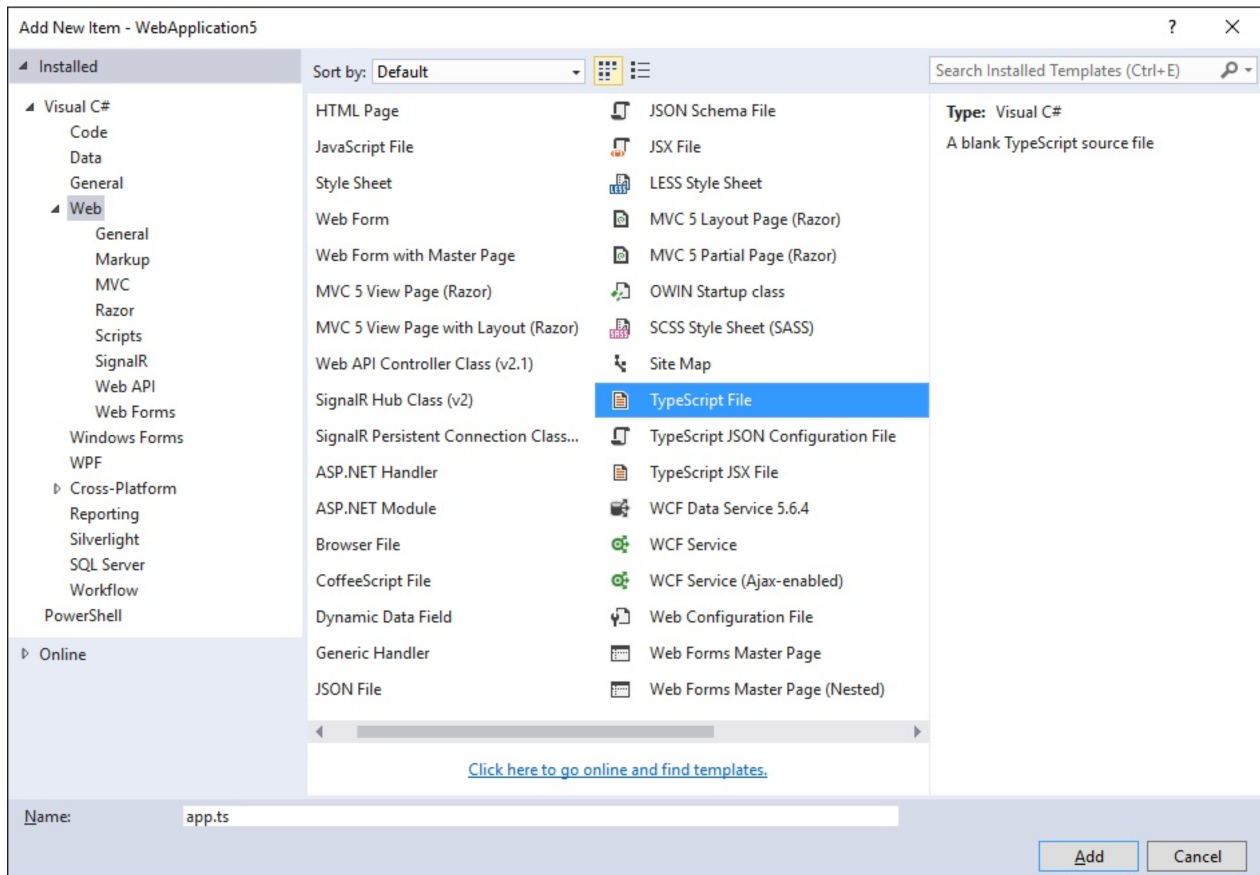

Application5\Project_Readme.html
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WebApplication5' (1 project)
Build
Rebuild
Clean
View
Analyze
Convert
Publish...
Configure Azure AD Authentication...
Add Application Insights Telemetry...
Scope to This
New Solution Explorer View
Show on Code Map
Add
Manage NuGet Packages...
Set as StartUp Project
Debug
Initialize Interactive with Project
Source Control
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Open Folder in File Explorer
Properties Alt+Enter
DEPLOY
Ensure your app is ready for
Area...
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Scaffolded Item...
New Folder
Add ASP.NET Folder
Azure API App SDK...
Azure API App Client...
New Azure WebJob Project
Existing Project as Azure WebJob
Reference...
Service Reference...
Connected Service...
Analyzer...
TypeScript File
TypeScript JSX File
HTML Page
JavaScript File
Style Sheet
Web Form
Class...

将文件夹命名为 src。



添加 TypeScript 代码

在 `src` 上右击并选择 **New Item**。接着选择 **TypeScript File** 并将此文件命名为 `app.ts` 。



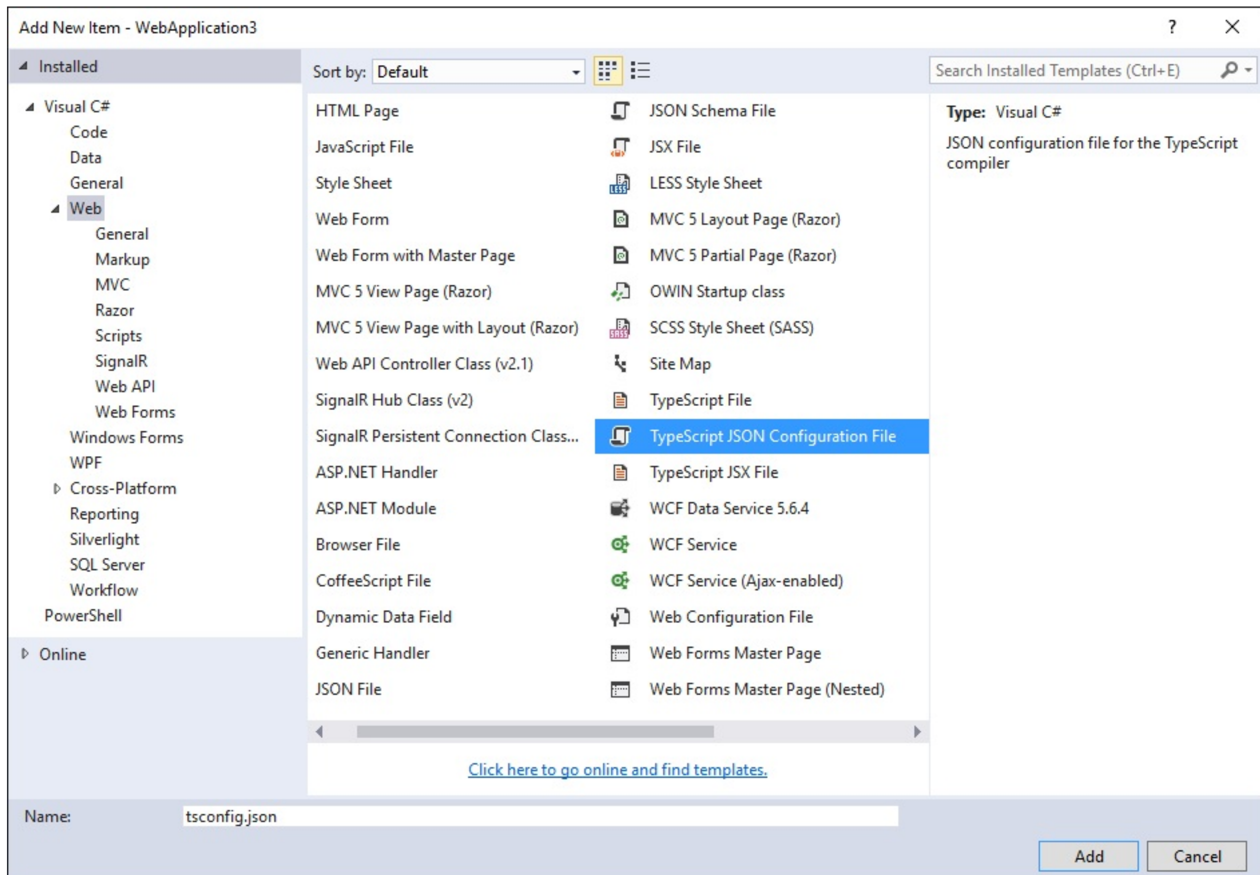
添加示例代码

将以下代码写入 `app.ts` 文件。

```
function sayHello() {
    const compiler = (document.getElementById("compiler") as HTMLInputElement).value;
    const framework = (document.getElementById("framework") as HTMLInputElement).value;
    return `Hello from ${compiler} and ${framework}!`;
}
```

构建设置

右击项目并选择 **New Item**。接着选择 **TypeScript Configuration File** 保持文件的默认名字为 `tsconfig.json` 。



将默认的 `tsconfig.json` 内容改为如下所示：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5",
    "outDir": "./Scripts/App"
  },
  "files": [
    "./src/app.ts",
  ],
  "compileOnSave": true
}
```

看起来和默认的设置差不多，但注意以下不同之处：

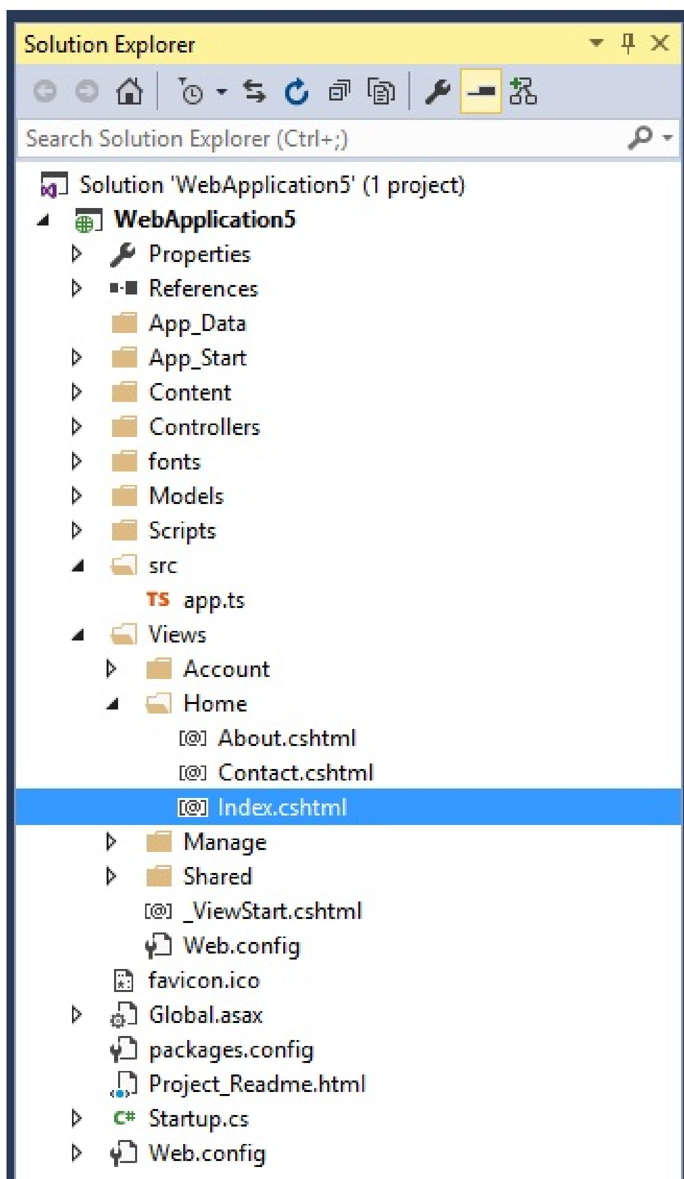
1. 设置 `"noImplicitAny": true` 。
2. 特别是这里 `"outDir": "./Scripts/App"` 。
3. 显式列出了 `"files"` 而不是依据 `"excludes"` 选项。

4. 设置 `"compileOnSave": true` 。

当你写新代码时，设置 `"noImplicitAny"` 选项是个好主意 — 这可以确保你不会错写任何新的类型。设置 `"compileOnSave"` 选项可以确保你在运行web程序前自动编译保存变更后的代码。更多信息请参见 the tsconfig.json documentation。

在视图中调用脚本

1. 在 **Solution Explorer** 中，打开 **Views | Home |** `Index.cshtml` 。

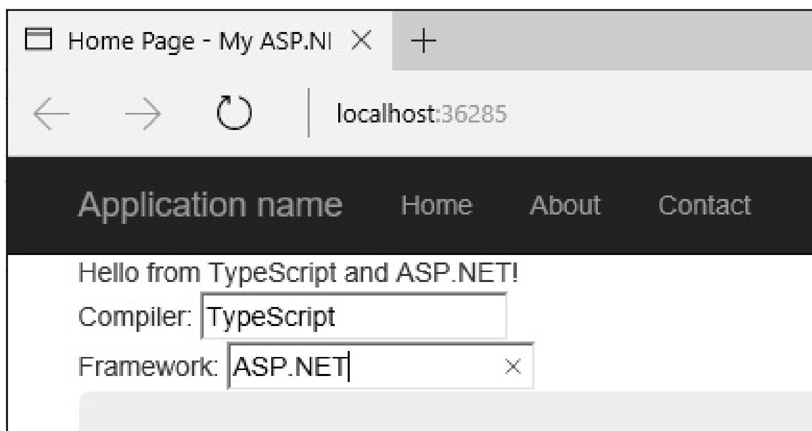


2. 修改代码如下：

```
@{
    ViewBag.Title = "Home Page";
}
<script src="~/Scripts/App/app.js"></script>
<div id="message"></div>
<div>
    Compiler: <input id="compiler" value="TypeScript" onkeyup
="document.getElementById('message').innerText = sayHello()"
 /><br />
    Framework: <input id="framework" value="ASP.NET" onkeyup=
"document.getElementById('message').innerText = sayHello()"
/>
</div>
```

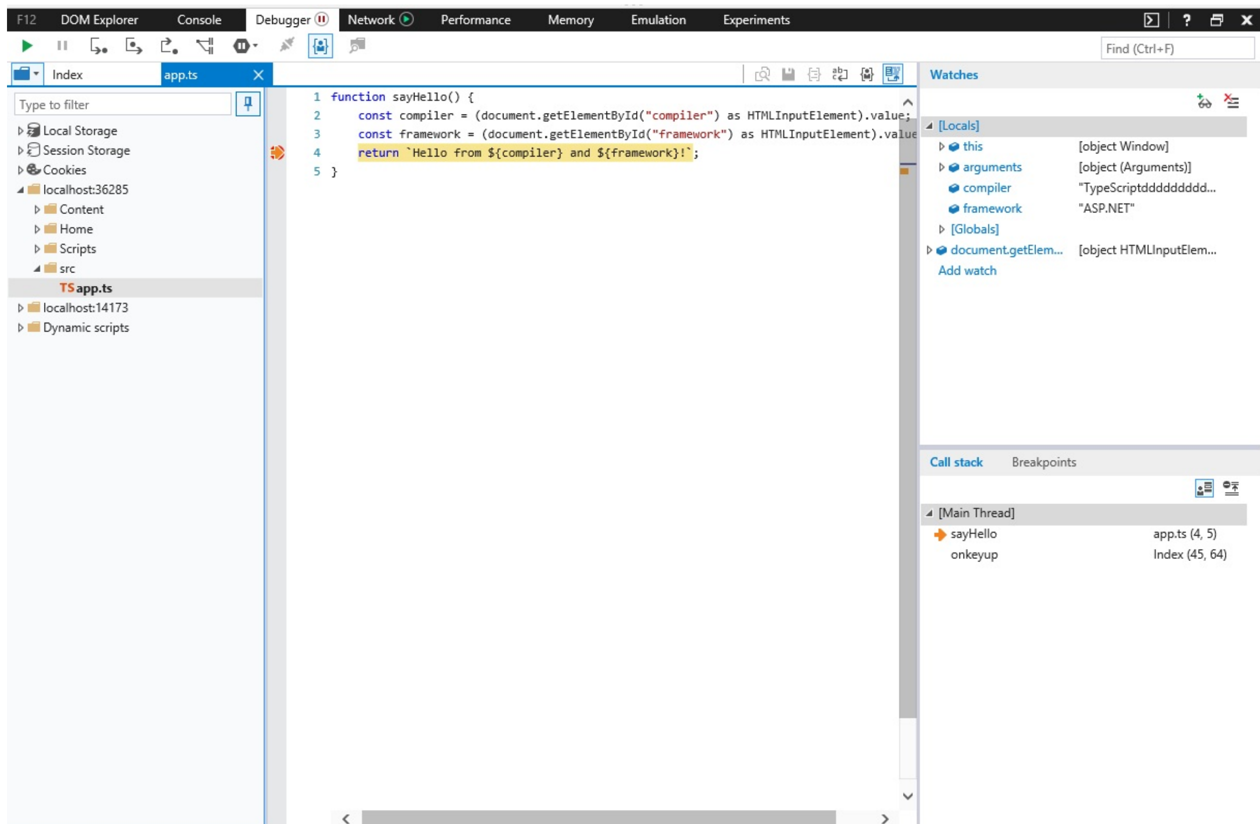
测试

1. 运行项目。
2. 在输入框中键入时，您应该看到一个消息：



调试

1. 在 Edge 浏览器中，按 F12 键并选择 **Debugger** 标签页。
2. 展开 localhost 列表，选择 src/app.ts
3. 在 `return` 那一行上打一个断点。
4. 在输入框中键入一些内容，确认TypeScript代码命中断点，观察它是否能正确地工作。



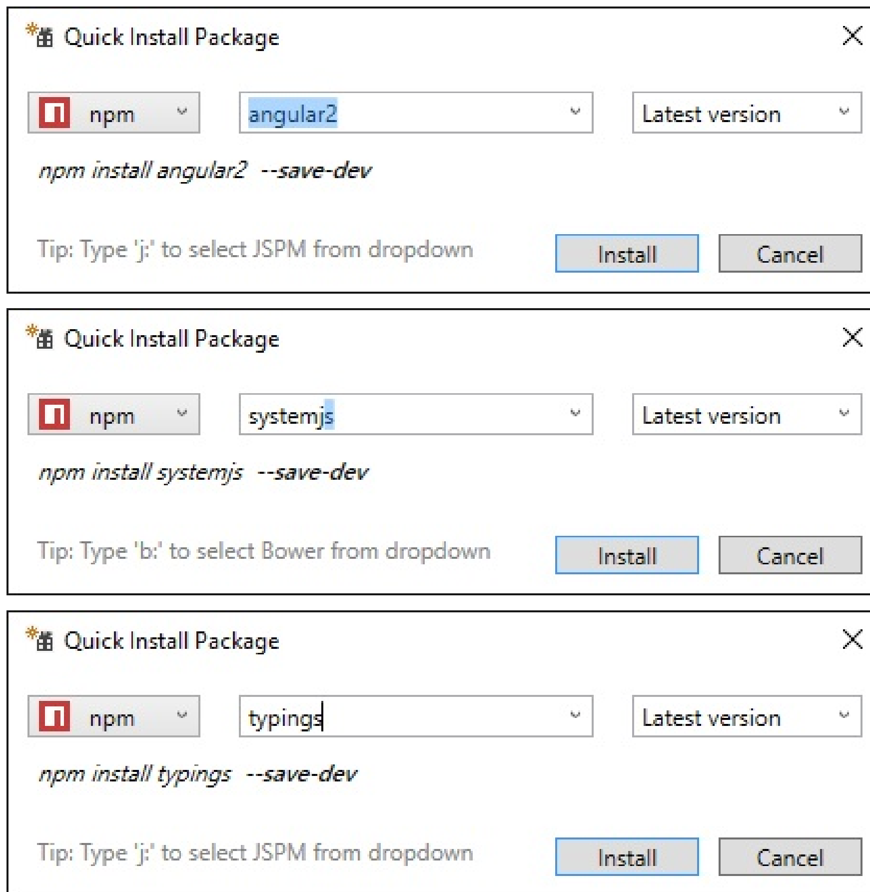
这就是你需要知道的在ASP.NET中使用TypeScript的基本知识了。接下来，我们引入Angular，写一个简单的Angular程序示例。

添加 Angular 2

使用 NPM 下载所需的包

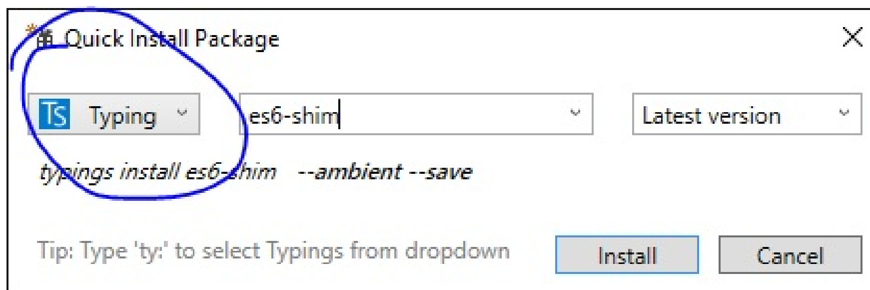
1. 安装 PackageInstaller。
2. 用 PackageInstaller 来安装 Angular 2， systemjs 和 Typings。

   在project上右击, 选择 **Quick Install Package**。



3. 用 PackageInstaller 安装 es6-shim 的类型文件。

   Angular 2 包含 es6-shim 以提供 Promise 支持, 但 TypeScript 还需要它的类型文件。 在 PackageInstaller 中, 选择 Typing 替换 npm 选项。接着键入 "es6-shim" :



更新 tsconfig.json

现在安装好了 Angular 2 及其依赖项， 我们还需要启用 TypeScript 中实验性的装饰器支持并且引入 es6-shim 的类型文件。 将来的版本中，装饰器和 ES6 选项将成为默认选项，我们就可以不做此设置了。 添加 `"experimentalDecorators": true,` `"emitDecoratorMetadata": true` 选项到 `"compilerOptions"` ，再添

加 `"./typings/index.d.ts"` 到 `"files"` 。 最后，我们要新建 `"./src/model.ts"` 文件，并且得把它加到 `"files"` 里。 现在 `tsconfig.json` 应该是这样：

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "sourceMap": true,
    "target": "es5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "outDir": "./Scripts/App"
  },
  "files": [
    "./src/app.ts",
    "./src/model.ts",
    "./src/main.ts",
    "./typings/index.d.ts"
  ]
}
```

添加 CopyFiles 到 build 中

最后，我们需要确保 Angular 文件作为 build 的一部分复制进来。这样操作，右击项目选择 'Unload' ，再次右击项目选择 'Edit csproj'。 在 TypeScript 配置项 PropertyGroup 之后，添加一个 ItemGroup 和 Target 配置项来复制 Angular 文件。

```
<ItemGroup>
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\angu
lar2\bundles\angular2.js"/>
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\angu
lar2\bundles\angular2-polyfills.js"/>
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\syst
emjs\dist\system.src.js"/>
  <NodeLib Include="$(MSBuildProjectDirectory)\node_modules\rxjs
\bundles\Rx.js"/>
</ItemGroup>
<Target Name="CopyFiles" BeforeTargets="Build">
  <Copy SourceFiles="@(NodeLib)" DestinationFolder="$(MSBuildPro
jectDirectory)\Scripts"/>
</Target>
```

现在，在工程上右击选择重新加载项目。此时应当能在解决方案资源管理器（Solution Explorer）中看到 `node_modules` 。

用 TypeScript 写一个简单的 Angular 应用

首先，将 `app.ts` 改成：

```
import {Component} from "angular2/core"
import {MyModel} from "./model"

@Component({
    selector: `my-app`,
    template: `<div>Hello from {{getCompiler()}}</div>`
})
class MyApp {
    model = new MyModel();
    getCompiler() {
        return this.model.compiler;
    }
}
```

接着在 `src` 中添加 TypeScript 文件 `model.ts` :

```
export class MyModel {
    compiler = "TypeScript";
}
```

再在 `src` 中添加 `main.ts` ：

```
import {bootstrap} from "angular2/platform/browser";
import {MyApp} from "./app";
bootstrap(MyApp);
```

最后，将 `Views/Home/Index.cshtml` 改成：

```
@{
    ViewBag.Title = "Home Page";
}
<script src="~/Scripts/angular2-polyfills.js"></script>
<script src="~/Scripts/system.src.js"></script>
<script src="~/Scripts/rx.js"></script>
<script src="~/Scripts/angular2.js"></script>
<script>
    System.config({
        packages: {
            '/Scripts/App': {
                format: 'cjs',
                defaultExtension: 'js'
            }
        }
    });
    System.import('/Scripts/App/main').then(null, console.error.
bind(console));
</script>
<my-app>Loading...</my-app>
```

这里加载了此应用。 运行 ASP.NET 应用，你应该能看到一个 div 显示 "Loading..." 紧接着更新成显示 "Hello from TypeScript"。

这篇快速上手指南将教你如何使用Gulp构建TypeScript，和如何在Gulp管道里添加Browserify，uglify或Watchify。 它还包涵了Babel的功能，通过使用Babelify。

这里假设你已经在使用Node.js和npm了。

创建简单工程

我们首先创建一个新目录。 命名为 `proj` ，也可以使用任何你喜欢的名字。

```
mkdir proj
cd proj
```

我们将以下面的结构开始我们的工程：

```
proj/
   ├─ src/
   └─ dist/
```

TypeScript文件放在 `src` 文件夹下，经过TypeScript编译器编译生成的目标文件放在 `dist` 目录下。

下面让我们来创建这些文件夹：

```
mkdir src
mkdir dist
```

初始化工程

现在让我们把这个文件夹转换成npm包：

```
npm init
```

你将看到有一些提示操作。 除了入口文件外，其余的都可以使用默认项。 入口文件使用 `./dist/main.js` 。 你可以随时在 `package.json` 文件里更改生成的配置。

安装依赖项

现在我们可以使用 `npm install` 命令来安装包。首先全局安装 `gulp-cli` （如果你使用Unix系统，你可能需要在 `npm install` 命令上使用 `sudo` ）。

```
npm install -g gulp-cli
```

然后安装 `typescript` ， `gulp` 和 `gulp-typescript` 到开发依赖项。Gulp-typescript是TypeScript的一个Gulp插件。

```
npm install --save-dev typescript gulp gulp-typescript
```

写一个简单的例子

让我们写一个Hello World程序。在 `src` 目录下创建 `main.ts` 文件：

```
function hello(compiler: string) {
    console.log(`Hello from ${compiler}`);
}
hello("TypeScript");
```

在工程的根目录 `proj` 下新建一个 `tsconfig.json` 文件：

```
{
    "files": [
        "src/main.ts"
    ],
    "compilerOptions": {
        "noImplicitAny": true,
        "target": "es5"
    }
}
```

新建 `gulpfile.js` 文件

在工程根目录下，新建一个 `gulpfile.js` 文件：

```
var gulp = require("gulp");
var ts = require("gulp-typescript");
var tsProject = ts.createProject("tsconfig.json");

gulp.task("default", function () {
    return tsProject.src()
        .pipe(tsProject())
        .js.pipe(gulp.dest("dist"));
});
```

测试这个应用

```
gulp
node dist/main.js
```

程序应该能够打印出“Hello from TypeScript!”。

向代码里添加模块

在使用Browserify前，让我们先构建一下代码然后再添加一些混入的模块。 这个结构将是你在真实应用程序中会用到的。

新建一个 `src/greet.ts` 文件：

```
export function sayHello(name: string) {
    return `Hello from ${name}`;
}
```

更改 `src/main.ts` 代码，从 `greet.ts` 导入 `sayHello` ：

```
import { sayHello } from "./greet";

console.log(sayHello("TypeScript"));
```

最后，将 `src/greet.ts` 添加到 `tsconfig.json` ：

```
{
    "files": [
        "src/main.ts",
        "src/greet.ts"
    ],
    "compilerOptions": {
        "noImplicitAny": true,
        "target": "es5"
    }
}
```

确保执行 `gulp` 后模块是能工作的，在Node.js下进行测试：

```
gulp
node dist/main.js
```

注意，即使我们使用了ES2015的模块语法，TypeScript还是会生成Node.js使用的CommonJS模块。 我们在这个教程里会一直使用CommonJS模块，但是你可以通过修改 `module` 选项来改变这个行为。

Browserify

现在，让我们把这个工程由Node.js环境移到浏览器环境里。 因此，我们将把所有模块捆绑成一个JavaScript文件。 所幸，这正是Browserify要做的事情。 更方便的是，它支持Node.js的CommonJS模块，这也正是TypeScript默认生成的类型。 也就是说TypeScript和Node.js的设置不需要改变就可以移植到浏览器里。

首先，安装Browserify，tsify和vinyl-source-stream。 tsify是Browserify的一个插件，就像gulp-typescript一样，它能够访问TypeScript编译器。 vinyl-source-stream会将Browserify的输出文件适配成gulp能够解析的格式，它叫做vinyl。

```
npm install --save-dev browserify tsify vinyl-source-stream
```

新建一个页面

在 `src` 目录下新建一个 `index.html` 文件：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Hello World!</title>
    </head>
    <body>
        <p id="greeting">Loading ...</p>
        <script src="bundle.js"></script>
    </body>
</html>
```

修改 `main.ts` 文件来更新这个页面：

```
import { sayHello } from "./greet";

function showHello(divName: string, name: string) {
    const elt = document.getElementById(divName);
    elt.innerText = sayHello(name);
}

showHello("greeting", "TypeScript");
```

`showHello` 调用 `sayHello` 函数更改页面上段落的文字。现在修改gulpfile文件如下：

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var tsify = require("tsify");
var paths = {
    pages: ['src/*.html']
};

gulp.task("copy-html", function () {
    return gulp.src(paths.pages)
        .pipe(gulp.dest("dist"));
});

gulp.task("default", ["copy-html"], function () {
    return browserify({
        basedir: '.',
        debug: true,
        entries: ['src/main.ts'],
        cache: {},
        packageCache: {}
    })
    .plugin(tsify)
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulp.dest("dist"));
});
```

这里增加了 `copy-html` 任务并且把它加作 `default` 的依赖项。这样，当 `default` 执行时， `copy-html` 会被首先执行。我们还修改了 `default` 任务，让它使用 `tsify` 插件调用Browserify，而不是 `gulp-typescript` 。方便的是，两者传递相同的参数对象到TypeScript编译器。

调用 `bundle` 后，我们使用 `source` （vinyl-source-stream的别名）把输出文件命名为 `bundle.js` 。

测试此页面，运行 `gulp` ，然后在浏览器里打开 `dist/index.html` 。你应该能在页面上看到“Hello from TypeScript”。

注意，我们为Broswerify指定了 `debug: true` 。 这会让 `tsify` 在输出文件里生成 `source maps` 。 `source maps` 允许我们在浏览器中直接调试TypeScript源码，而不是在合并后的JavaScript文件上调试。 你要打开调试器并在 `main.ts` 里打一个断点，看看 `source maps` 是否能工作。 当你刷新页面时，代码会停在断点处，从而你就能够调试 `greet.ts` 。

Watchify，Babel和Uglify

现在代码已经用Browserify和tsify捆绑在一起了，我们可以使用Browserify插件为构建添加一些特性。

- Watchify启动Gulp并保持运行状态，当你保存文件时自动编译。 帮你进入到编辑-保存-刷新浏览器的循环中。
- Babel是个十分灵活的编译器，将ES2015及以上版本的代码转换成ES5和ES3。 你可以添加大量自定义的TypeScript目前不支持的转换器。
- Uglify帮你压缩代码，将花费更少的时间去下载它们。

Watchify

我们启动Watchify，让它在后台帮我们编译：

```
npm install --save-dev watchify gulp-util
```

修改gulpfile文件如下：

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var watchify = require("watchify");
var tsify = require("tsify");
var gutil = require("gulp-util");
var paths = {
    pages: ['src/*.html']
};

var watchedBrowserify = watchify(browserify({
    basedir: '.',
    debug: true,
    entries: ['src/main.ts'],
    cache: {},
    packageCache: {}
}).plugin(tsify));

gulp.task("copy-html", function () {
    return gulp.src(paths.pages)
        .pipe(gulp.dest("dist"));
});

function bundle() {
    return watchedBrowserify
        .bundle()
        .pipe(source('bundle.js'))
        .pipe(gulp.dest("dist"));
}

gulp.task("default", ["copy-html"], bundle);
watchedBrowserify.on("update", bundle);
watchedBrowserify.on("log", gutil.log);
```

共有三处改变，但是需要你略微重构一下代码。

1. 将 `browserify` 实例包裹在 `watchify` 的调用里，控制生成的结果。
2. 调用 `watchedBrowserify.on("update", bundle);` ，每次TypeScript文件改变时Browserify会执行 `bundle` 函数。

3. 调用 `watchedBrowserify.on("log", gutil.log);` 将日志打印到控制台。

(1)和(2)在一起意味着我们要将 `browserify` 调用移出 `default` 任务。 然后给函数起个名字，因为Watchify和Gulp都要调用它。 (3)是可选的，但是对于调试来讲很有用。

现在当你执行 `gulp` ，它会启动并保持运行状态。 试着改变 `main.ts` 文件里 `showHello` 的代码并保存。 你会看到这样的输出：

```
proj$ gulp
[10:34:20] Using gulpfile ~/src/proj/gulpfile.js
[10:34:20] Starting 'copy-html'...
[10:34:20] Finished 'copy-html' after 26 ms
[10:34:20] Starting 'default'...
[10:34:21] 2824 bytes written (0.13 seconds)
[10:34:21] Finished 'default' after 1.36 s
[10:35:22] 2261 bytes written (0.02 seconds)
[10:35:24] 2808 bytes written (0.05 seconds)
```

Uglify

首先安装Uglify。 因为Uglify是用于混淆你的代码，所以我们还要安装vinyl-buffer和gulp-sourcemaps来支持sourcemaps。

```
npm install --save-dev gulp-uglify vinyl-buffer gulp-sourcemaps
```

修改gulpfile文件如下：

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require('vinyl-source-stream');
var tsify = require("tsify");
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var buffer = require('vinyl-buffer');
var paths = {
    pages: ['src/*.html']
};

gulp.task("copy-html", function () {
    return gulp.src(paths.pages)
        .pipe(gulp.dest("dist"));
});

gulp.task("default", ["copy-html"], function () {
    return browserify({
        basedir: '.',
        debug: true,
        entries: ['src/main.ts'],
        cache: {},
        packageCache: {}
    })
    .plugin(tsify)
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
    .pipe(uglify())
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest("dist"));
});
```

注意 `uglify` 只是调用了自己— `buffer` 和 `sourcemaps` 的调用是用于确保sourcemaps可以工作。 这些调用让我们可以使用单独的sourcemap文件，而不是之前的内嵌的sourcemaps。 你现在可以执行 `gulp` 来检查 `bundle.js` 是否被压缩了：

```
gulp
cat dist/bundle.js
```

Babel

首先安装Babelify和ES2015的Babel预置程序。 和Uglify一样，Babelify也会混淆代码，因此我们也需要vinyl-buffer和gulp-sourcemaps。 默认情况下Babelify只会处理扩展名为 `.js` ， `.es` ， `.es6` 和 `.jsx` 的文件，因此我们需要添加 `.ts` 扩展名到Babelify选项。

```
npm install --save-dev babelify babel-core babel-preset-es2015 v
inyl-buffer gulp-sourcemaps
```

修改gulpfile文件如下：

```
var gulp = require('gulp');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var tsify = require('tsify');
var sourcemaps = require('gulp-sourcemaps');
var buffer = require('vinyl-buffer');
var paths = {
    pages: ['src/*.html']
};

gulp.task('copyHtml', function () {
    return gulp.src(paths.pages)
        .pipe(gulp.dest('dist'));
});

gulp.task('default', ['copyHtml'], function () {
    return browserify({
        basedir: '.',
        debug: true,
        entries: ['src/main.ts'],
        cache: {},
        packageCache: {}
    })
    .plugin(tsify)
    .transform('babelify', {
        presets: ['es2015'],
        extensions: ['.ts']
    })
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('dist'));
});
```

我们需要设置TypeScript目标为ES2015。 Babel稍后会从TypeScript生成的ES2015代码中生成ES5。 修改 `tsconfig.json` :

```
{
    "files": [
        "src/main.ts"
    ],
    "compilerOptions": {
        "noImplicitAny": true,
        "target": "es2015"
    }
}
```

对于这样一段简单的代码来说，Babel的ES5输出应该和TypeScript的输出相似。

> 注意： 此教程已从官方删除

这个快速上手指南会告诉你如何结合使用TypeScript和Knockout.js。

这里我们假设你已经会使用Node.js和npm

新建工程

首先，我们新建一个目录。 暂时命名为 `proj` ，当然了你可以使用任何喜欢的名字。

```
mkdir proj
cd proj
```

接下来，我们按如下方式来组织这个工程：

```
proj/
   ├─ src/
   └─ built/
```

TypeScript源码放在 `src` 目录下，结过TypeScript编译器编译后，生成的文件放在 `built` 目录里。

下面创建目录：

```
mkdir src
mkdir built
```

初始化工程

现在将这个文件夹转换为npm包。

```
npm init
```

你会看到一系列提示。除了入口点外其它设置都可以使用默认值。你可以随时到生成的 `package.json` 文件里修改这些设置。

安装构建依赖

首先确保TypeScript已经全局安装。

```
npm install -g typescript
```

我们还要获取Knockout的声明文件，它描述了这个库的结构供TypeScript使用。

```
npm install --save @types/knockout
```

获取运行时依赖

我们需要Knockout和RequireJS。RequireJS是一个库，它可以让我们在运行时异步地加载模块。

有以下几种获取方式：

1. 手动下载文件并维护它们。
2. 通过像Bower这样的包管理下载并维护它们。
3. 使用内容分发网络（CDN）来维护这两个文件。

我们使用第一种方法，它会简单一些，但是Knockout的官方文档上有讲解如何使用CDN，更多像RequireJS一样的代码库可以在cdnjs上查找。

下面让我们在工程根目录下创建 `externals` 目录。

```
mkdir externals
```

然后下载Knockout和下载RequireJS到这个目录里。最新的压缩后版本就可以。

添加TypeScript配置文件

下面我们想把所有的TypeScript文件整合到一起 - 包括自己写的和必须的声明文件。

我们需要创建一个 `tsconfig.json` 文件，包含了输入文件列表和编译选项。 在工程根目录下创建一个新文件 `tsconfig.json` ，内容如下：

```
{
    "compilerOptions": {
        "outDir": "./built/",
        "sourceMap": true,
        "noImplicitAny": true,
        "module": "amd",
        "target": "es5"
    },
    "files": [
        "./src/require-config.ts",
        "./src/hello.ts"
    ]
}
```

这里引用了 `typings/index.d.ts` ，它是Typings帮我们创建的。 这个文件会自动地包含所有安装的依赖。

你可能会对 `typings` 目录下的 `browser.d.ts` 文件感到好奇，尤其因为我们将在浏览器里运行代码。 其实原因是这样的，当目标为浏览器的时候，一些包会生成不同的版本。 通常来讲，这些情况很少发生并且在这里我们不会遇到这种情况，所以我们可以忽略 `browser.d.ts` 。

你可以在这里查看更多关于 `tsconfig.json` 文件的信息

写些代码

下面我们使用Knockout写一段TypeScript代码。 首先，在 `src` 目录里新建一个 `hello.ts` 文件。

```
import * as ko from "knockout";

class HelloViewModel {
    language: KnockoutObservable<string>
    framework: KnockoutObservable<string>

    constructor(language: string, framework: string) {
        this.language = ko.observable(language);
        this.framework = ko.observable(framework);
    }
}

ko.applyBindings(new HelloViewModel("TypeScript", "Knockout"));
```

接下来，在 `src` 目录下再新建一个 `require-config.ts` 文件。

```
declare var require: any;
require.config({
    paths: {
        "knockout": "externals/knockout-3.4.0",
    }
});
```

这个文件会告诉RequireJS从哪里导入Knockout，好比我们在 `hello.ts` 里做的一样。 你创建的所有页面都应该在RequireJS之后和导入任何东西之前引入它。 为了更好地理解这个文件和如何配置RequireJS，可以查看文档。

我们还需要一个视图来显示 `HelloViewModel` 。 在 `proj` 目录的根上创建一个文件 `index.html` ，内容如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Hello Knockout!</title>
    </head>
    <body>
        <p>
            Hello from
            <strong data-bind="text: language">todo</strong>
            and
            <strong data-bind="text: framework">todo</strong>!
        </p>

        <p>Language: <input data-bind="value: language" /></p>
        <p>Framework: <input data-bind="value: framework" /></p>

        <script src="./externals/require.js"></script>
        <script src="./built/require-config.js"></script>
        <script>
            require(["built/hello"]);
        </script>
    </body>
</html>
```

注意，有两个script标签。 首先，我们引入RequireJS。 然后我们再在 `require-config.js` 里映射外部依赖，这样RequireJS就能知道到哪里去查找它们。 最后，使用我们要去加载的模块去调用 `require` 。

将所有部分整合在一起

运行

```
tsc
```

现在，在你喜欢的浏览器打开 `index.html` ，所有都应该好用了。 你应该可以看到页面上显示“Hello from TypeScript and Knockout!”。 在它下面，你还会看到两个输入框。 当你改变输入和切换焦点时，就会看到原先显示的信息改变了。

这篇指南将会教你如何将TypeScript和React还有webpack结合在一起使用。

如果你正在做一个全新的工程，可以先阅读这篇React快速上手指南。

否则，我们假设已经在使用Node.js和npm。

初始化项目结构

让我们新建一个目录。将会命名为 `proj`，但是你可以改成任何你喜欢的名字。

```
mkdir proj
cd proj
```

我们会像下面的结构组织我们的工程：

```
proj/
├─ dist/
└─ src/
   └─ components/
```

TypeScript文件会放在 `src` 文件夹里，通过TypeScript编译器编译，然后经webpack处理，最后生成一个 `bundle.js` 文件放在 `dist` 目录下。我们自定义的组件将会放在 `src/components` 文件夹下。

下面来创建基本结构：

```
mkdir src
cd src
mkdir components
cd ..
```

Webpack会帮助我们生成 `dist` 目录。

初始化工程

现在把这个目录变成npm包。

```
npm init
```

你会看到一些提示，放心地使用默认值就可以了。 当然，你也可以随时到生成的 `package.json` 文件里修改。

安装依赖

首先确保已经全局安装了Webpack。

```
npm install -g webpack
```

Webpack这个工具可以将你的所有代码和可选择地将依赖捆绑成一个单独的 `.js` 文件。

现在我们添加React和React-DOM以及它们的声明文件到 `package.json` 文件里做为依赖：

```
npm install --save react react-dom @types/react @types/react-dom
```

使用 `@types/` 前缀表示我们额外要获取React和React-DOM的声明文件。 通常当你导入像 `"react"` 这样的路径，它会查看 `react` 包； 然而，并不是所有的包都包含了声明文件，所以TypeScript还会查看 `@types/react` 包。 你会发现我们以后将不必在意这些。

接下来，我们要添加开发时依赖awesome-typescript-loader和source-map-loader。

```
npm install --save-dev typescript awesome-typescript-loader sour
ce-map-loader
```

这些依赖会让TypeScript和webpack在一起良好地工作。 awesome-typescript-loader可以让Webpack使用TypeScript的标准配置文件 `tsconfig.json` 编译TypeScript代码。 source-map-loader使用TypeScript输出的sourcemap文件来告诉webpack何时生成自己的sourcemaps。 这就允许你在调试最终生成的文件时就好像在调试TypeScript源码一样。

注意我们安装TypeScript为一个开发依赖。 我们还可以使用 `npm link typescript` 来链接TypeScript到一个全局拷贝，但这不是常见用法。

添加TypeScript配置文件

我们想将TypeScript文件整合到一起 - 这包括我们写的源码和必要的声明文件。

我们需要创建一个 `tsconfig.json` 文件，它包含了输入文件列表以及编译选项。在工程根目录下新建文件 `tsconfig.json` 文件，添加以下内容：

```
{
    "compilerOptions": {
        "outDir": "./dist/",
        "sourceMap": true,
        "noImplicitAny": true,
        "module": "commonjs",
        "target": "es5",
        "jsx": "react"
    },
    "include": [
        "./src/**/*"
    ]
}
```

你可以在这里了解更多关于 `tsconfig.json` 文件的说明。

写些代码

下面使用React写一段TypeScript代码。 首先，在 `src/components` 目录下创建一个名为 `Hello.tsx` 的文件，代码如下：

```
import * as React from "react";

export interface HelloProps { compiler: string; framework: strin
g; }

export const Hello = (props: HelloProps) => <h1>Hello from {prop
s.compiler} and {props.framework}!</h1>;
```

注意这个例子使用了无状态的功能组件，我们可以让它更像一点类。

```
import * as React from "react";

export interface HelloProps { compiler: string; framework: strin
g; }

// 'HelloProps' describes the shape of props.
// State is never set so we use the '{}' type.
export class Hello extends React.Component<HelloProps, {}> {
    render() {
        return <h1>Hello from {this.props.compiler} and {this.pr
ops.framework}!</h1>;
    }
}
```

接下来，在 `src` 下创建 `index.tsx` 文件，源码如下：

```
import * as React from "react";
import * as ReactDOM from "react-dom";

import { Hello } from "./components/Hello";

ReactDOM.render(
    <Hello compiler="TypeScript" framework="React" />,
    document.getElementById("example")
);
```

我们仅仅将 `Hello` 组件导入 `index.tsx` 。 注意，不同于 `"react"` 或 `"react-dom"` ，我们使用 `Hello.tsx` 的相对路径 - 这很重要。 如果不这样做，TypeScript只会尝试在 `node_modules` 文件夹里查找。

我们还需要一个页面来显示 `Hello` 组件。 在根目录 `proj` 创建一个名为 `index.html` 的文件，如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Hello React!</title>
    </head>
    <body>
        <div id="example"></div>

        <!-- Dependencies -->
        <script src="./node_modules/react/umd/react.development.
js"></script>
        <script src="./node_modules/react-dom/umd/react-dom.deve
lopment.js"></script>

        <!-- Main -->
        <script src="./dist/bundle.js"></script>
    </body>
</html>
```

需要注意一点我们是从 `node_modules` 引入的文件。 React和React-DOM的npm包里包含了独立的 `.js` 文件，你可以在页面上引入它们，这里我们为了快捷就直接引用了。 可以随意地将它们拷贝到其它目录下，或者从CDN上引用。 Facebook在CND上提供了一系列可用的React版本，你可以在这里查看更多内容。

创建一个webpack配置文件

在工程根目录下创建一个 `webpack.config.js` 文件。

```
module.exports = {
```

```
    entry: "./src/index.tsx",
    output: {
        filename: "bundle.js",
        path: __dirname + "/dist"
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: [".ts", ".tsx", ".js", ".json"]
    },

    module: {
        rules: [
            // All files with a '.ts' or '.tsx' extension will b
e handled by 'awesome-typescript-loader'.
            { test: /\.tsx?$/, loader: "awesome-typescript-loade
r" },

            // All output '.js' files will have any sourcemaps r
e-processed by 'source-map-loader'.
            { enforce: "pre", test: /\.js$/, loader: "source-map
-loader" }
        ]
    },

    // When importing a module whose path matches one of the fol
lowing, just
    // assume a corresponding global variable exists and use tha
t instead.
    // This is important because it allows us to avoid bundling
all of our
    // dependencies, which allows browsers to cache those librar
ies between builds.
    externals: {
        "react": "React",
        "react-dom": "ReactDOM"
    }
```

```
};
```

大家可能对 `externals` 字段有所疑惑。 我们想要避免把所有的React都放到一个文件里，因为会增加编译时间并且浏览器还能够缓存没有发生改变的库文件。

理想情况下，我们只需要在浏览器里引入React模块，但是大部分浏览器还没有支持模块。 因此大部分代码库会把自己包裹在一个单独的全局变量内，比如： `jQuery` 或 `_` 。 这叫做“命名空间”模式，webpack也允许我们继续使用通过这种方式写的代码库。 通过我们的设置 `"react": "React"` ，webpack会神奇地将所有对 `"react"` 的导入转换成从 `React` 全局变量中加载。

你可以在这里了解更多如何配置webpack。

整合在一起

执行：

```
webpack
```

在浏览器里打开 `index.html` ，工程应该已经可以用了！ 你可以看到页面上显示着“Hello from TypeScript and React!”

这篇快速上手指南会教你如何将TypeScript与React结合起来使用。 在最后，你将学到：

- 使用TypeScript和React创建工程
- 使用TSLint进行代码检查
- 使用Jest和Enzyme进行测试，以及
- 使用Redux管理状态

我们会使用create-react-app工具快速搭建工程环境。

这里假设你已经在使用Node.js和npm。 并且已经了解了React的基础知识。

安装create-react-app

我们之所以使用create-react-app是因为它能够为React工程设置一些有效的工具和权威的默认参数。 它仅仅是一个用来搭建React工程的命令行工具而已。

```
npm install -g create-react-app
```

创建新工程

让我们首先创建一个叫做 `my-app` 的新工程：

```
create-react-app my-app --scripts-version=react-scripts-ts
```

react-scripts-ts是一系列适配器，它利用标准的create-react-app工程管道并把TypeScript混入进来。

此时的工程结构应如下所示：

```
my-app/
├─ .gitignore
├─ node_modules/
├─ public/
├─ src/
|  └─ ...
├─ package.json
├─ tsconfig.json
└─ tslint.json
```

注意：

- `tsconfig.json` 包含了工程里TypeScript特定的选项。
- `tslint.json` 保存了要使用的代码检查器的设置，TSLint。
- `package.json` 包含了依赖，还有一些命令的快捷方式，如测试命令，预览命令和发布应用的命令。
- `public` 包含了静态资源如HTML页面或图片。除了 `index.html` 文件外，其它的文件都可以删除。
- `src` 包含了TypeScript和CSS源码。 `index.tsx` 是强制使用的入口文件。

运行工程

通过下面的方式即可轻松地运行这个工程。

```
npm run start
```

它会执行 `package.json` 里面指定的 `start` 命令，并且会启动一个服务器，当我们保存文件时还会自动刷新页面。 通常这个服务器的地址是 `http://localhost:3000` ，页面应用会被自动地打开。

它会保持监听以方便我们快速地预览改动。

测试工程

测试也仅仅是一行命令的事儿：

```
npm run test
```

这个命令会运行Jest，一个非常好用的测试工具，它会运行所有扩展名是 `.test.ts` 或 `.spec.ts` 的文件。 好比是 `npm run start` 命令，当检测到有改动的时候Jest会自动地运行。 如果喜欢的话，你还可以同时运行 `npm run start` 和 `npm run test` ，这样你就可以在预览的同时进行测试。

生成生产环境的构建版本

在使用 `npm run start` 运行工程的时候，我们并没有生成一个优化过的版本。 通常我们想给用户一个运行的尽可能快并在体积上尽可能小的代码。 像压缩这样的优化方法可以做到这一点，但是总是要耗费更多的时间。 我们把这样的构建版本称做"生产环境"版本（与开发版本相对）。

要执行生产环境的构建，可以运行如下命令：

```
npm run build
```

这会相应地创建优化过的JS和CSS文件， `./build/static/js` 和 `./build/static/css` 。

大多数情况下你不需要生成生产环境的构建版本， 但它可以帮助你衡量应用最终版本的体积大小。

创建一个组件

下面我们将要创建一个 `Hello` 组件。 这个组件接收任意一个我们想对之打招呼的名字（我们把它叫做 `name` ），并且有一个可选数量的感叹号做为结尾（通过 `enthusiasmLevel` ）。

若我们这样写 `<Hello name="Daniel" enthusiasmLevel={3} />` ，这个组件大至会渲染成 `<div>Hello Daniel!!!</div>` 。 如果没指定 `enthusiasmLevel` ，组件将默认显示一个感叹号。 若 `enthusiasmLevel` 为 `0` 或负值将抛出一个错误。

下面来写一下 `Hello.tsx` ：

```
// src/components/Hello.tsx

import * as React from 'react';

export interface Props {
  name: string;
  enthusiasmLevel?: number;
}

function Hello({ name, enthusiasmLevel = 1 }: Props) {
  if (enthusiasmLevel <= 0) {
    throw new Error('You could be a little more enthusiastic. :D
');
  }

  return (
    <div className="hello">
      <div className="greeting">
        Hello {name + getExclamationMarks(enthusiasmLevel)}
      </div>
    </div>
  );
}

export default Hello;

// helpers

function getExclamationMarks(numChars: number) {
  return Array(numChars + 1).join('!');
}
```

注意我们定义了一个类型 `Props` ，它指定了我们组件要用到的属性。 `name` 是必需的且为 `string` 类型，同时 `enthusiasmLevel` 是可选的且为 `number` 类型（你可以通过名字后面加 `?` 为指定可选参数）。

我们创建了一个无状态的函数式组件（Stateless Functional Components，SFC）`Hello`。具体来讲，`Hello`是一个函数，接收一个`Props`对象并拆解它。如果`Props`对象里没有设置`enthusiasmLevel`，默认值为`1`。

使用函数是React中定义组件的两种方式之一。如果你喜欢的话，也可以通过类的方式定义：

```
class Hello extends React.Component<Props, object> {
  render() {
    const { name, enthusiasmLevel = 1 } = this.props;

    if (enthusiasmLevel <= 0) {
      throw new Error('You could be a little more enthusiastic.
:D');
    }

    return (
      <div className="hello">
        <div className="greeting">
          Hello {name + getExclamationMarks(enthusiasmLevel)}
        </div>
      </div>
    );
  }
}
```

当我们的组件具有某些状态的时候，使用类的方式是很有用处的。但在这个例子里我们不需要考虑状态 - 事实上，在`React.Component<Props, object>`我们把状态指定为了`object`，因此使用SFC更简洁。当在创建可重用的通用UI组件的时候，在表现层使用组件局部状态比较适合。针对我们应用的生命周期，我们会审视应用是如何通过Redux轻松地管理普通状态的。

现在我们已经写好了组件，让我们仔细看看`index.tsx`，把`<App />`替换成`<Hello ... />`。

首先我们在文件头部导入它：

```
import Hello from './components/Hello';
```

然后修改 `render` 调用：

```
ReactDOM.render(
  <Hello name="TypeScript" enthusiasmLevel={10} />,
  document.getElementById('root') as HTMLElement
);
```

类型断言

这里还有一点要指出，就是最后一行 `document.getElementById('root') as HTMLElement` 。这个语法叫做类型断言，有时也叫做转换。当你比类型检查器更清楚一个表达式的类型的时候，你可以通过这种方式通知TypeScript。

这里，我们之所以这么做是因为 `getElementById` 的返回值类型是 `HTMLElement | null` 。简单地说， `getElementById` 返回 `null` 是当无法找对对应 `id` 元素的时候。我们假设 `getElementById` 总是成功的，因此我们要使用 `as` 语法告诉TypeScript这点。

TypeScript还有一种感叹号（ `!` ）结尾的语法，它会从前面的表达式里移除 `null` 和 `undefined` 。所以我们也可以写成 `document.getElementById('root')!` ，但在这里我们想写的更清楚些。

:sunglasses:添加样式

通过我们的设置为一个组件添加样式很容易。若要设置 `Hello` 组件的样式，我们可以创建这样一个CSS文件 `src/components/Hello.css` 。

```
.hello {
  text-align: center;
  margin: 20px;
  font-size: 48px;
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif
}

.hello button {
    margin-left: 25px;
    margin-right: 25px;
    font-size: 40px;
    min-width: 50px;
}
```

`create-react-app` 包含的工具（Webpack和一些加载器）允许我们导入样式表文件。 当我们构建应用的时候，所有导入的 `.css` 文件会被拼接成一个输出文件。 因此在 `src/components/Hello.tsx` ，我们需要添加如下导入语句。

```
import './Hello.css';
```

使用Jest编写测试

如果你没使用过Jest，你可能先要把它安装为开发依赖项。

```
npm install -D jest jest-cli jest-config
```

我们对 `Hello` 组件有一些假设。 让我们在此重申一下：

> - 当这样写 `<Hello name="Daniel" enthusiasmLevel={3} />` 时，组件应被渲染成 `<div>Hello Daniel!!!</div>` 。
> - 若未指定 `enthusiasmLevel` ，组件应默认显示一个感叹号。
> - 若 `enthusiasmLevel` 为 `0` 或负值，它应抛出一个错误。

我们将针对这些需求为组件写一些注释。

但首先，我们要安装Enzyme。 Enzyme是React生态系统里一个通用工具，它方便了针对组件的行为编写测试。 默认地，我们的应用包含了一个叫做jsdom的库，它允许我们模拟DOM以及在非浏览器的环境下测试运行时的行为。 Enzyme与此类似，但是是基于jsdom的，并且方便我们查询组件。

让我们把它安装为开发依赖项。

```
npm install -D enzyme @types/enzyme enzyme-adapter-react-16 @typ
es/enzyme-adapter-react-16
```

如果你的react版本低于15.5.0，还需安装如下

```
npm install -D react-addons-test-utils
```

注意我们同时安装了 `enzyme` 和 `@types/enzyme` 。 `enzyme` 包指的是包含了实际运行的JavaScript代码包，而 `@types/enzyme` 则包含了声明文件（ `.d.ts` 文件）的包，以便TypeScript能够了解该如何使用Enzyme。 你可以在这里了解更多关于 `@types` 包的信息。

我们还需要安装 `enzyme-adapter` 和 `react-addons-test-utils` 。 它们是使用 `enzyme` 所需要安装的包，前者作为配置适配器是必须的，而后者若采用的React版本在15.5.0之上则毋需安装。

现在我们已经设置好了Enzyme，下面开始编写测试！ 先创建一个文件 `src/components/Hello.test.tsx` ，与先前的 `Hello.tsx` 文件放在一起。

```
// src/components/Hello.test.tsx

import * as React from 'react';
import * as enzyme from 'enzyme';
import * as Adapter from 'enzyme-adapter-react-16';
import Hello from './Hello';

enzyme.configure({ adapter: new Adapter() });

it('renders the correct text when no enthusiasm level is given',
 () => {
  const hello = enzyme.shallow(<Hello name='Daniel' />);
```

```
  expect(hello.find(".greeting").text()).toEqual('Hello Daniel!')
});

it('renders the correct text with an explicit enthusiasm of 1', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={1}/>);
  expect(hello.find(".greeting").text()).toEqual('Hello Daniel!')
});

it('renders the correct text with an explicit enthusiasm level of 5', () => {
  const hello = enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={5} />);
  expect(hello.find(".greeting").text()).toEqual('Hello Daniel!!!!!');
});

it('throws when the enthusiasm level is 0', () => {
  expect(() => {
    enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={0} />);
  }).toThrow();
});

it('throws when the enthusiasm level is negative', () => {
  expect(() => {
    enzyme.shallow(<Hello name='Daniel' enthusiasmLevel={-1} />);
  }).toThrow();
});
```

这些测试都十分基础，但你可以从中得到启发。

添加state管理

到此为止，如果你使用React的目的是只获取一次数据并显示，那么你已经完成了。 但是如果你想开发一个可以交互的应用，那么你需要添加state管理。

state管理概述

React本身就是一个适合于创建可组合型视图的库。 但是，React并没有任何在应用间同步数据的功能。 就React组件而言，数据是通过每个元素上指定的props向子元素传递。

因为React本身并没有提供内置的state管理功能，React社区选择了Redux和MobX库。

Redux依靠一个统一且不可变的数据存储来同步数据，并且更新那里的数据时会触发应用的更新渲染。 state的更新是以一种不可变的方式进行，它会发布一条明确的action消息，这个消息必须被reducer函数处理。 由于使用了这样明确的方式，很容易弄清楚一个action是如何影响程序的state。

MobX借助于函数式响应型模式，state被包装在了可观察对象里，并通过props传递。 通过将state标记为可观察的，即可在所有观察者之间保持state的同步性。 另一个好处是，这个库已经使用TypeScript实现了。

这两者各有优缺点。 但Redux使用得更广泛，因此在这篇教程里，我们主要看如何使用Redux； 但是也鼓励大家两者都去了解一下。

后面的小节学习曲线比较陡。 因此强烈建议大家先去熟悉一下Redux。

设置actions

只有当应用里的state会改变的时候，我们才需要去添加Redux。 我们需要一个action的来源，它将触发改变。 它可以是一个定时器或者UI上的一个按钮。

为此，我们将增加两个按钮来控制 `Hello` 组件的感叹级别。

安装Redux

安装 `redux` 和 `react-redux` 以及它们的类型文件做为依赖。

```
npm install -S redux react-redux @types/react-redux
```

这里我们不需要安装 `@types/redux` ，因为Redux已经自带了声明文件（ `.d.ts` 文件）。

定义应用的状态

我们需要定义Redux保存的state的结构。 创建 `src/types/index.tsx` 文件，它保存了类型的定义，我们在整个程序里都可能用到。

```
// src/types/index.tsx

export interface StoreState {
    languageName: string;
    enthusiasmLevel: number;
}
```

这里我们想让 `languageName` 表示应用使用的编程语言（例如，TypeScript或者JavaScript）， `enthusiasmLevel` 是可变的。 在写我们的第一个容器的时候，就会明白为什么要令state与props稍有不同。

添加actions

下面我们创建这个应用将要响应的消息类型， `src/constants/index.tsx` 。

```
// src/constants/index.tsx

export const INCREMENT_ENTHUSIASM = 'INCREMENT_ENTHUSIASM';
export type INCREMENT_ENTHUSIASM = typeof INCREMENT_ENTHUSIASM;


export const DECREMENT_ENTHUSIASM = 'DECREMENT_ENTHUSIASM';
export type DECREMENT_ENTHUSIASM = typeof DECREMENT_ENTHUSIASM;
```

这里的 `const` / `type` 模式允许我们以容易访问和重构的方式使用TypeScript的字符串字面量类型。

接下来，我们创建一些actions以及创建这些actions的函数， `src/actions/index.tsx` 。

```
import * as constants from '../constants'

export interface IncrementEnthusiasm {
    type: constants.INCREMENT_ENTHUSIASM;
}

export interface DecrementEnthusiasm {
    type: constants.DECREMENT_ENTHUSIASM;
}

export type EnthusiasmAction = IncrementEnthusiasm | DecrementEn
thusiasm;

export function incrementEnthusiasm(): IncrementEnthusiasm {
    return {
        type: constants.INCREMENT_ENTHUSIASM
    }
}

export function decrementEnthusiasm(): DecrementEnthusiasm {
    return {
        type: constants.DECREMENT_ENTHUSIASM
    }
}
```

我们创建了两个类型，它们负责增加操作和减少操作的行为。 我们还定义了一个类型（ `EnthusiasmAction` ），它描述了哪些action是可以增加或减少的。 最后，我们定义了两个函数用来创建实际的actions。

这里有一些清晰的模版，你可以参考类似redux-actions的库。

添加reducer

现在我们可以开始写第一个reducer了！ Reducers是函数，它们负责生成应用state的拷贝使之产生变化，但它并没有副作用。 它们是一种纯函数。

我们的reducer将放在 `src/reducers/index.tsx` 文件里。 它的功能是保证增加操作会让感叹级别加1，减少操作则要将感叹级别减1，但是这个级别永远不能小于1。

```
// src/reducers/index.tsx

import { EnthusiasmAction } from '../actions';
import { StoreState } from '../types/index';
import { INCREMENT_ENTHUSIASM, DECREMENT_ENTHUSIASM } from '../constants/index';

export function enthusiasm(state: StoreState, action: EnthusiasmAction): StoreState {
  switch (action.type) {
    case INCREMENT_ENTHUSIASM:
      return { ...state, enthusiasmLevel: state.enthusiasmLevel + 1 };
    case DECREMENT_ENTHUSIASM:
      return { ...state, enthusiasmLevel: Math.max(1, state.enthusiasmLevel - 1) };
  }
  return state;
}
```

注意我们使用了对象展开（ `...state` ），当替换 `enthusiasmLevel` 时，它可以对状态进行浅拷贝。 将 `enthusiasmLevel` 属性放在末尾是十分关键的，否则它将被旧的状态覆盖。

你可能想要对reducer写一些测试。 因为reducers是纯函数，它们可以传入任意的数据。 针对每个输入，可以测试reducers生成的新的状态。 可以考虑使用Jest的toEqual方法。

创建容器

在使用Redux时，我们常常要创建组件和容器。 组件是数据无关的，且工作在表现层。 容器通常包裹组件及其使用的数据，用以显示和修改状态。 你可以在这里阅读更多关于这个概念的细节：Dan Abramov写的表现层的容器组件。

现在我们修改 `src/components/Hello.tsx` ，让它可以修改状态。 我们将添加两个可选的回调属性到 `Props` ，它们分别是 `onIncrement` 和 `onDecrement` ：

```
export interface Props {
  name: string;
  enthusiasmLevel?: number;
  onIncrement?: () => void;
  onDecrement?: () => void;
}
```

然后将这两个回调绑定到两个新按钮上，将按钮添加到我们的组件里。

```
function Hello({ name, enthusiasmLevel = 1, onIncrement, onDecre
ment }: Props) {
  if (enthusiasmLevel <= 0) {
    throw new Error('You could be a little more enthusiastic. :D
');
  }

  return (
    <div className="hello">
      <div className="greeting">
        Hello {name + getExclamationMarks(enthusiasmLevel)}
      </div>
      <div>
        <button onClick={onDecrement}>-</button>
        <button onClick={onIncrement}>+</button>
      </div>
    </div>
  );
}
```

通常情况下，我们应该给 `onIncrement` 和 `onDecrement` 写一些测试，它们是在各自的按钮被点击时调用。 试一试以便掌握编写测试的窍门。

现在我们的组件更新好了，可以把它放在一个容器里了。 让我们来创建一个文件 `src/containers/Hello.tsx` ，在开始的地方使用下列导入语句。

```
import Hello from '../components/Hello';
import * as actions from '../actions/';
import { StoreState } from '../types/index';
import { connect, Dispatch } from 'react-redux';
```

两个关键点是初始的 `Hello` 组件和react-redux的 `connect` 函数。 `connect` 可以将我们的 `Hello` 组件转换成一个容器，通过以下两个函数：

- `mapStateToProps` 将当前store里的数据以我们的组件需要的形式传递到组件。
- `mapDispatchToProps` 利用 `dispatch` 函数，创建回调props将actions送到store。

回想一下，我们的应用包含两个属性： `languageName` 和 `enthusiasmLevel` 。我们的 `Hello` 组件，希望得到一个 `name` 和一个 `enthusiasmLevel` 。 `mapStateToProps` 会从store得到相应的数据，如果需要的话将针对组件的props调整它。 下面让我们继续往下写。

```
export function mapStateToProps({ enthusiasmLevel, languageName }: StoreState) {
  return {
    enthusiasmLevel,
    name: languageName,
  }
}
```

注意 `mapStateToProps` 仅创建了 `Hello` 组件需要的四个属性中的两个。 我们还想要传入 `onIncrement` 和 `onDecrement` 回调函数。 `mapDispatchToProps` 是一个函数，它需要传入一个调度函数。 这个调度函数可以将actions传入store来触发更新，因此我们可以创建一对回调函数，它们会在需要的时候调用调度函数。

```
export function mapDispatchToProps(dispatch: Dispatch<actions.EnthusiasmAction>) {
  return {
    onIncrement: () => dispatch(actions.incrementEnthusiasm()),
    onDecrement: () => dispatch(actions.decrementEnthusiasm()),
  }
}
```

最后，我们可以调用 `connect` 了。 `connect` 首先会接收 `mapStateToProps` 和 `mapDispatchToProps` ，然后返回另一个函数，我们用它来包裹我们的组件。最终的容器是通过下面的代码定义的：

```
export default connect(mapStateToProps, mapDispatchToProps)(Hello);
```

现在，我们的文件应该是下面这个样子：

```
// src/containers/Hello.tsx

import Hello from '../components/Hello';
import * as actions from '../actions/';
import { StoreState } from '../types/index';
import { connect, Dispatch } from 'react-redux';

export function mapStateToProps({ enthusiasmLevel, languageName
}: StoreState) {
  return {
    enthusiasmLevel,
    name: languageName,
  }
}

export function mapDispatchToProps(dispatch: Dispatch<actions.En
thusiasmAction>) {
  return {
    onIncrement: () => dispatch(actions.incrementEnthusiasm()),
    onDecrement: () => dispatch(actions.decrementEnthusiasm()),
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Hell
o);
```

创建store

让我们回到 `src/index.tsx` 。要把所有的东西合到一起，我们需要创建一个带初始状态的store，并用我们所有的reducers来设置它。

```
import { createStore } from 'redux';
import { enthusiasm } from './reducers/index';
import { StoreState } from './types/index';

const store = createStore<StoreState>(enthusiasm, {
  enthusiasmLevel: 1,
  languageName: 'TypeScript',
});
```

`store` 可能正如你想的那样，它是我们应用全局状态的核心store。

接下来，我们将要用 `./src/containers/Hello` 来包裹 `./src/components/Hello` ，然后使用react-redux的 `Provider` 将props与容器连通起来。 我们将导入它们：

```
import Hello from './containers/Hello';
import { Provider } from 'react-redux';
```

将 `store` 以 `Provider` 的属性形式传入：

```
ReactDOM.render(
  <Provider store={store}>
    <Hello />
  </Provider>,
  document.getElementById('root') as HTMLElement
);
```

注意， `Hello` 不再需要props了，因为我们使用了 `connect` 函数为包裹起来的 `Hello` 组件的props适配了应用的状态。

退出

如果你发现create-react-app使一些自定义设置变得困难，那么你就可以选择不使用它，使用你需要配置。 比如，你要添加一个Webpack插件，你就可以利用create-react-app提供的“eject”功能。

运行：

```
npm run eject
```

这样就可以了！

你要注意，在运行eject前最好保存你的代码。 你不能撤销eject命令，因此退出操作是永久性的除非你从一个运行eject前的提交来恢复工程。

下一步

create-react-app带有很多很棒的功能。 它们的大多数都在我们工程生成的 `README.md` 里面有记录，所以可以简单阅读一下。

如果你想学习更多关于Redux的知识，你可以前往官方站点查看文档。 同样的，MobX官方站点。

如果你想要在某个时间点eject，你需要了解再多关于Webpack的知识。 你可以查看React & Webpack教程。

有时候你需要路由功能。 已经有一些解决方案了，但是对于Redux工程来讲react-router是最流行的，并经常与react-router-redux联合使用。

即将到来的Angular 2框架是使用TypeScript开发的。 因此Angular和TypeScript一起使用非常简单方便。 Angular团队也在其文档里把TypeScript视为一等公民。

正因为这样，你总是可以在Angular 2官网(或Angular 2官网中文版)里查看到最新的结合使用Angular和TypeScript的参考文档。 在这里查看快速上手指南，现在就开始学习吧！

TypeScript不是存在于真空中。 它从JavaScript生态系统和大量现存的JavaScript而来。 将JavaScript代码转换成TypeScript虽乏味却不是难事。 接下来这篇教程将教你怎么做。 在开始转换TypeScript之前，我们假设你已经理解了足够多本手册里的内容。

如果你打算要转换一个React工程，推荐你先阅读React转换指南。

设置目录

如果你在写纯JavaScript，你大概是想直接运行这些JavaScript文件， 这些文件存在于 `src` ， `lib` 或 `dist` 目录里，它们可以按照预想运行。

若如此，那么你写的纯JavaScript文件将做为TypeScript的输入，你将要运行的是TypeScript的输出。 在从JS到TS的转换过程中，我们会分离输入文件以防TypeScript覆盖它们。 你也可以指定输出目录。

你可能还需要对JavaScript做一些中间处理，比如合并或经过Babel再次编译。 在这种情况下，你应该已经有了如下的目录结构。

那么现在，我们假设你已经设置了这样的目录结构：

```
projectRoot
├── src
│   ├── file1.js
│   └── file2.js
├── built
└── tsconfig.json
```

如果你在 `src` 目录外还有 `tests` 文件夹，那么在 `src` 里可以有一个 `tsconfig.json` 文件，在 `tests` 里还可以有一个。

书写配置文件

TypeScript使用 `tsconfig.json` 文件管理工程配置，例如你想包含哪些文件和进行哪些检查。 让我们先创建一个简单的工程配置文件：

```
{
    "compilerOptions": {
        "outDir": "./built",
        "allowJs": true,
        "target": "es5"
    },
    "include": [
        "./src/**/*"
    ]
}
```

这里我们为TypeScript设置了一些东西:

1. 读取所有可识别的 `src` 目录下的文件（通过 `include` ）。
2. 接受JavaScript做为输入（通过 `allowJs` ）。
3. 生成的所有文件放在 `built` 目录下（通过 `outDir` ）。
4. 将JavaScript代码降级到低版本比如ECMAScript 5（通过 `target` ）。

现在，如果你在工程根目录下运行 `tsc` ，就可以在 `built` 目录下看到生成的文件。 `built` 下的文件应该与 `src` 下的文件相同。 现在你的工程里的TypeScript已经可以工作了。

早期收益

现在你已经可以看到TypeScript带来的好处，它能帮助我们理解当前工程。 如果你打开像VS Code或Visual Studio这样的编译器，你就能使用像自动补全这样的工具。 你还可以配置如下的选项来帮助查找BUG：

- `noImplicitReturns` 会防止你忘记在函数末尾返回值。
- `noFallthroughCasesInSwitch` 会防止在 `switch` 代码块里的两个 `case` 之间忘记添加 `break` 语句。

TypeScript还能发现那些执行不到的代码和标签，你可以通过设置 `allowUnreachableCode` 和 `allowUnusedLabels` 选项来禁用。

与构建工具进行集成

在你的构建管道中可能包含多个步骤。 比如为每个文件添加一些内容。 每种工具的使用方法都是不同的，我们会尽可能的包涵主流的工具。

Gulp

如果你在使用时髦的Gulp，我们已经有一篇关于使用Gulp结合TypeScript并与常见构建工具Browserify，Babelify和Uglify进行集成的教程。 请阅读这篇教程。

Webpack

Webpack集成非常简单。 你可以使用 `awesome-typescript-loader` ，它是一个TypeScript的加载器，结合 `source-map-loader` 方便调试。 运行：

```
npm install awesome-typescript-loader source-map-loader
```

并将下面的选项合并到你的 `webpack.config.js` 文件里：

```
module.exports = {
    entry: "./src/index.ts",
    output: {
        filename: "./dist/bundle.js",
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'awesome-typescript-loader'.
            { test: /\.tsx?$/, loader: "awesome-typescript-loader" }
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
            { test: /\.js$/, loader: "source-map-loader" }
        ]
    },

    // Other options...
};
```

要注意的是，`awesome-typescript-loader` 必须在其它处理 `.js` 文件的加载器之前运行。

这与另一个TypeScript的Webpack加载器ts-loader是一样的。你可以到这里了解两者之间的差别。

你可以在React和Webpack教程里找到使用Webpack的例子。

转换到TypeScript文件

到目前为止，你已经做好了使用TypeScript文件的准备。 第一步，将 `.js` 文件重命名为 `.ts` 文件。 如果你使用了JSX，则重命名为 `.tsx` 文件。

第一步达成？ 太棒了! 你已经成功地将一个文件从JavaScript转换成了TypeScript!

当然了，你可能感觉哪里不对劲儿。 如果你在支持TypeScript的编辑器（或运行 `tsc --pretty` ）里打开了那个文件，你可能会看到有些行上有红色的波浪线。 你可以把它们当做在Microsoft Word里看到的红色波浪线一样。 但是TypeScript仍然会编译你的代码，就好比Word还是允许你打印你的文档一样。

如果对你来说这种行为太随便了，你可以让它变得严格些。 如果，你不想在发生错误的时候，TypeScript还会被编译成JavaScript，你可以使用 `noEmitOnError` 选项。 从某种意义上来讲，TypeScript具有一个调整它的严格性的刻度盘，你可以将指针拔动到你想要的位置。

如果你计划使用可用的高度严格的设置，最好现在就启用它们（查看启用严格检查）。 比如，如果你不想让TypeScript将没有明确指定的类型默默地推断为 `any` 类型，可以在修改文件之前启用 `noImplicitAny` 。 你可能会觉得这有些过度严格，但是长期收益很快就能显现出来。

去除错误

我们提到过，若不出所料，在转换后将会看到错误信息。 重要的是我们要逐一的查看它们并决定如何处理。 通常这些都是真正的BUG，但有时必须要告诉TypeScript你要做的是什么。

由模块导入

首先你可能会看到一些类似 `Cannot find name 'require'.` 和 `Cannot find name 'define'.` 的错误。 遇到这种情况说明你在使用模块。 你仅需要告诉TypeScript它们是存在的：

```
// For Node/CommonJS
declare function require(path: string): any;
```

或

```
// For RequireJS/AMD
declare function define(...args: any[]): any;
```

最好是避免使用这些调用而改用TypeScript的导入语法。

首先，你要使用TypeScript的 `module` 标记来启用一些模块系统。 可用的选项有 `commonjs` ， `amd` ， `system` ，and `umd` 。

如果代码里存在下面的Node/CommonJS代码：

```
var foo = require("foo");

foo.doStuff();
```

或者下面的RequireJS/AMD代码：

```
define(["foo"], function(foo) {
    foo.doStuff();
})
```

那么可以写做下面的TypeScript代码：

```
import foo = require("foo");

foo.doStuff();
```

获取声明文件

如果你开始做转换到TypeScript导入，你可能会遇到 `Cannot find module 'foo'.` 这样的错误。 问题出在没有声明文件来描述你的代码库。 幸运的是这非常简单。 如果TypeScript报怨像是没有 `lodash` 包，那你只需这样做

```
npm install -S @types/lodash
```

如果你没有使用 `commonjs` 模块模块选项，那么就需要将 `moduleResolution` 选项设置为 `node` 。

之后，你应该就可以导入 `lodash` 了，并且会获得精确的自动补全功能。

由模块导出

通常来讲，由模块导出涉及添加属性到 `exports` 或 `module.exports` 。 TypeScript允许你使用顶级的导出语句。 比如，你要导出下面的函数：

```
module.exports.feedPets = function(pets) {
    // ...
}
```

那么你可以这样写：

```
export function feedPets(pets) {
    // ...
}
```

有时你会完全重写导出对象。 这是一个常见模式，这会将模块变为可立即调用的模块：

```
var express = require("express");
var app = express();
```

之前你可以是这样写的：

```
function foo() {
    // ...
}
module.exports = foo;
```

在TypeScript里，你可以使用 `export =` 来代替。

```
function foo() {
    // ...
}
export = foo;
```

过多或过少的参数

有时你会发现你在调用一个具有过多或过少参数的函数。 通常，这是一个BUG，但在某些情况下，你可以声明一个使用 `arguments` 对象的函数而不需要写出所有参数:

```
function myCoolFunction() {
    if (arguments.length == 2 && !Array.isArray(arguments[1])) {
        var f = arguments[0];
        var arr = arguments[1];
        // ...
    }
    // ...
}

myCoolFunction(function(x) { console.log(x) }, [1, 2, 3, 4]);
myCoolFunction(function(x) { console.log(x) }, 1, 2, 3, 4);
```

这种情况下，我们需要利用TypeScript的函数重载来告诉调用者 `myCoolFunction` 函数的调用方式。

```
function myCoolFunction(f: (x: number) => void, nums: number[]):
 void;
function myCoolFunction(f: (x: number) => void, ...nums: number[
]): void;
function myCoolFunction() {
    if (arguments.length == 2 && !Array.isArray(arguments[1])) {
        var f = arguments[0];
        var arr = arguments[1];
        // ...
    }
    // ...
}
```

我们为 `myCoolFunction` 函数添加了两个重载签名。第一个检查 `myCoolFunction` 函数是否接收一个函数（它又接收一个 `number` 参数）和一个 `number` 数组。第二个同样是接收了一个函数，并且使用剩余参数（ `...nums` ）来表示之后的其它所有参数必须是 `number` 类型。

连续添加属性

有些人可能会因为代码美观性而喜欢先创建一个对象然后立即添加属性：

```
var options = {};
options.color = "red";
options.volume = 11;
```

TypeScript会提示你不能给 `color` 和 `volumn` 赋值，因为先前指定 `options` 的类型为 `{}` 并不带有任何属性。如果你将声明变成对象字面量的形式将不会产生错误：

```
let options = {
    color: "red",
    volume: 11
};
```

你还可以定义 `options` 的类型并且添加类型断言到对象字面量上。

```
interface Options { color: string; volume: number }

let options = {} as Options;
options.color = "red";
options.volume = 11;
```

或者，你可以将 `options` 指定成 `any` 类型，这是最简单的，但也是获益最少的。

`any`，`Object`，和 `{}`

你可能会试图使用 `Object` 或 `{}` 来表示一个值可以具有任意属性，因为 `Object` 是最通用的类型。 然而在这种情况下 **`any`** 是真正想要使用的类型，因为它是最灵活的类型。

比如，有一个 `Object` 类型的东西，你将不能够在其上调用 `toLowerCase()` 。

越普通意味着更少的利用类型，但是 `any` 比较特殊，它是最普通的类型但是允许你在上面做任何事情。 也就是说你可以在上面调用，构造它，访问它的属性等等。 记住，当你使用 `any` 时，你会失去大多数TypeScript提供的错误检查和编译器支持。

如果你还是决定使用 `Object` 和 `{}` ，你应该选择 `{}` 。 虽说它们基本一样，但是从技术角度上来讲 `{}` 在一些深奥的情况里比 `Object` 更普通。

启用严格检查

TypeScript提供了一些检查来保证安全以及帮助分析你的程序。 当你将代码转换为了TypeScript后，你可以启用这些检查来帮助你获得高度安全性。

没有隐式的 `any`

在某些情况下TypeScript没法确定某些值的类型。 那么TypeScript会使用 `any` 类型代替。 这对代码转换来讲是不错，但是使用 `any` 意味着失去了类型安全保障，并且你得不到工具的支持。 你可以使用 `noImplicitAny` 选项，让TypeScript标记出发生这种情况的地方，并给出一个错误。

严格的 `null` 与 `undefined` 检查

默认地，TypeScript把 `null` 和 `undefined` 当做属于任何类型。 这就是说，声明为 `number` 类型的值可以为 `null` 和 `undefined` 。 因为在JavaScript和TypeScript里， `null` 和 `undefined` 经常会导致BUG的产生，所以TypeScript包含了 `strictNullChecks` 选项来帮助我们减少对这种情况的担忧。

当启用了 `strictNullChecks` ， `null` 和 `undefined` 获得了它们自己各自的类型 `null` 和 `undefined` 。 当任何值可能为 `null` ，你可以使用联合类型。 比如，某值可能为 `number` 或 `null` ，你可以声明它的类型为 `number | null` 。

假设有一个值TypeScript认为可以为 `null` 或 `undefined` ，但是你更清楚它的类型，你可以使用 `!` 后缀。

```
declare var foo: string[] | null;

foo.length;  // error - 'foo' is possibly 'null'

foo!.length; // okay - 'foo!' just has type 'string[]'
```

要当心，当你使用 `strictNullChecks` ，你的依赖也需要相应地启用 `strictNullChecks` 。

`this` 没有隐式的 `any`

当你在类的外部使用 `this` 关键字时，它会默认获得 `any` 类型。 比如，假设有一个 `Point` 类，并且我们要添加一个函数做为它的方法：

```
class Point {
    constructor(public x, public y) {}
    getDistance(p: Point) {
        let dx = p.x - this.x;
        let dy = p.y - this.y;
        return Math.sqrt(dx ** 2 + dy ** 2);
    }
}
// ...

// Reopen the interface.
interface Point {
    distanceFromOrigin(point: Point): number;
}
Point.prototype.distanceFromOrigin = function(point: Point) {
    return this.getDistance({ x: 0, y: 0});
}
```

这就产生了我们上面提到的错误 - 如果我们错误地拼写了 `getDistance` 并不会得到一个错误。正因此，TypeScript有 `noImplicitThis` 选项。当设置了它，TypeScript会产生一个错误当没有明确指定类型（或通过类型推断）的 `this` 被使用时。解决的方法是在接口或函数上使用指定了类型的 `this` 参数：

```
Point.prototype.distanceFromOrigin = function(this: Point, point
: Point) {
    return this.getDistance({ x: 0, y: 0});
}
```

Table of Contents

介绍

为了让程序有价值，我们需要能够处理最简单的数据单元：数字，字符串，结构体，布尔值等。 TypeScript支持与JavaScript几乎相同的数据类型，此外还提供了实用的枚举类型方便我们使用。

布尔值

最基本的数据类型就是简单的true/false值，在JavaScript和TypeScript里叫做 `boolean` （其它语言中也一样）。

```
let isDone: boolean = false;
```

数字

和JavaScript一样，TypeScript里的所有数字都是浮点数。 这些浮点数的类型是 `number` 。 除了支持十进制和十六进制字面量，TypeScript还支持ECMAScript 2015中引入的二进制和八进制字面量。

```
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
let binaryLiteral: number = 0b1010;
let octalLiteral: number = 0o744;
```

字符串

JavaScript程序的另一项基本操作是处理网页或服务器端的文本数据。 像其它语言里一样，我们使用 `string` 表示文本数据类型。 和JavaScript一样，可以使用双引号（ `"` ）或单引号（ `'` ）表示字符串。

```
let name: string = "bob";
name = "smith";
```

你还可以使用模版字符串，它可以定义多行文本和内嵌表达式。 这种字符串是被反引号包围（ ` ），并且以 `${ expr }` 这种形式嵌入表达式

```
let name: string = `Gene`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ name }.

I'll be ${ age + 1 } years old next month.`;
```

这与下面定义 `sentence` 的方式效果相同：

```
let sentence: string = "Hello, my name is " + name + ".\n\n" +
    "I'll be " + (age + 1) + " years old next month.";
```

数组

TypeScript像JavaScript一样可以操作数组元素。 有两种方式可以定义数组。 第一种，可以在元素类型后面接上 `[]` ，表示由此类型元素组成的一个数组：

```
let list: number[] = [1, 2, 3];
```

第二种方式是使用数组泛型， `Array<元素类型>` ：

```
let list: Array<number> = [1, 2, 3];
```

元组 Tuple

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。 比如，你可以定义一对值分别为 `string` 和 `number` 类型的元组。

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ['hello', 10]; // OK
// Initialize it incorrectly
x = [10, 'hello']; // Error
```

当访问一个已知索引的元素，会得到正确的类型：

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

当访问一个越界的元素，会使用联合类型替代：

```
x[3] = 'world'; // OK, 字符串可以赋值给(string | number)类型

console.log(x[5].toString()); // OK, 'string' 和 'number' 都有 toString

x[6] = true; // Error, 布尔不是(string | number)类型
```

联合类型是高级主题，我们会在以后的章节里讨论它。

枚举

`enum` 类型是对JavaScript标准数据类型的一个补充。 像C#等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

默认情况下，从 `0` 开始为元素编号。 你也可以手动的指定成员的数值。 例如，我们将上面的例子改成从 `1` 开始编号：

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

或者，全部都采用手动赋值：

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。 例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

alert(colorName);  // 显示'Green'因为上面代码里它的值是2
```

任意值

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。 这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。 这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。 那么我们可以使用 `any` 类型来标记这些变量：

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

在对现有代码进行改写的时候， `any` 类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。 你可能认为 `Object` 有相似的作用，就像它在其它语言中那样。 但是 `Object` 类型的变量只是允许你给它赋任意值 - 但是却不能够在它上面调用任意的方法，即便它真的有这些方法：

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

当你只知道一部分数据的类型时， `any` 类型也是有用的。 比如，你有一个数组，它包含了不同的类型的数据：

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

空值

某种程度上来说， `void` 类型像是与 `any` 类型相反，它表示没有任何类型。 当一个函数没有返回值时，你通常会见到其返回值类型是 `void` ：

```
function warnUser(): void {
    alert("This is my warning message");
}
```

声明一个 `void` 类型的变量没有什么大用，因为你只能为它赋予 `undefined` 和 `null` ：

```
let unusable: void = undefined;
```

Null 和 Undefined

TypeScript里， `undefined` 和 `null` 两者各自有自己的类型分别叫做 `undefined` 和 `null` 。 和 `void` 相似，它们的本身的类型用处不是很大：

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

默认情况下 `null` 和 `undefined` 是所有类型的子类型。 就是说你可以把 `null` 和 `undefined` 赋值给 `number` 类型的变量。

然而，当你指定了 `--strictNullChecks` 标记， `null` 和 `undefined` 只能赋值给 `void` 和它们各自。 这能避免很多常见的问题。 也许在某处你想传入一个 `string` 或 `null` 或 `undefined` ，你可以使用联合类型 `string | null | undefined` 。 再次说明，稍后我们会介绍联合类型。

> 注意：我们鼓励尽可能地使用 `--strictNullChecks` ，但在本手册里我们假设这个标记是关闭的。

Never

`never` 类型表示的是那些永不存在的值的类型。 例如， `never` 类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型；变量也可能是 `never` 类型，当它们被永不为真的类型保护所约束时。

`never` 类型是任何类型的子类型，也可以赋值给任何类型；然而，没有类型是 `never` 的子类型或可以赋值给 `never` 类型（除了 `never` 本身之外）。 即使 `any` 也不可以赋值给 `never` 。

下面是一些返回 `never` 类型的函数：

```
// 返回never的函数必须存在无法达到的终点
function error(message: string): never {
    throw new Error(message);
}

// 推断的返回值类型为never
function fail() {
    return error("Something failed");
}

// 返回never的函数必须存在无法达到的终点
function infiniteLoop(): never {
    while (true) {
    }
}
```

Object

`object` 表示非原始类型，也就是除 `number` ， `string` ， `boolean` ， `symbol` ， `null` 或 `undefined` 之外的类型。

使用 `object` 类型，就可以更好的表示像 `Object.create` 这样的API。例如：

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

类型断言

有时候你会遇到这样的情况，你会比TypeScript更了解某个值的详细信息。 通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。 类型断言好比其它语言里的类型转换，但是不进行特殊的数据检查和解构。 它没有运行时的影响，只是在编译阶段起作用。 TypeScript会假设你，程序员，已经进行了必须的检查。

类型断言有两种形式。 其一是“尖括号”语法：

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

另一个为 `as` 语法：

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

两种形式是等价的。 至于使用哪个大多数情况下是凭个人喜好；然而，当你在TypeScript里使用JSX时，只有 `as` 语法断言是被允许的。

关于 `let`

你可能已经注意到了，我们使用 `let` 关键字来代替大家所熟悉的JavaScript关键字 `var` 。 `let` 关键字是JavaScript的一个新概念，TypeScript实现了它。 我们会在以后详细介绍它，很多常见的问题都可以通过使用 `let` 来解决，所以尽可能地使用 `let` 来代替 `var` 吧。

变量声明

`let` 和 `const` 是JavaScript里相对较新的变量声明方式。 像我们之前提到过的， `let` 在很多方面与 `var` 是相似的，但是可以帮助大家避免在JavaScript里常见一些问题。 `const` 是对 `let` 的一个增强，它能阻止对一个变量再次赋值。

因为TypeScript是JavaScript的超集，所以它本身就支持 `let` 和 `const` 。 下面我们会详细说明这些新的声明方式以及为什么推荐使用它们来代替 `var` 。

如果你之前使用JavaScript时没有特别在意，那么这节内容会唤起你的回忆。 如果你已经对 `var` 声明的怪异之处了如指掌，那么你可以轻松地略过这节。

`var` 声明

一直以来我们都是通过 `var` 关键字定义JavaScript变量。

```
var a = 10;
```

大家都能理解，这里定义了一个名为 `a` 值为 `10` 的变量。

我们也可以在函数内部定义变量：

```
function f() {
    var message = "Hello, world!";

    return message;
}
```

并且我们也可以在其它函数内部访问相同的变量。

```
function f() {
    var a = 10;
    return function g() {
        var b = a + 1;
        return b;
    }
}

var g = f();
g(); // returns 11;
```

上面的例子里，`g` 可以获取到 `f` 函数里定义的 `a` 变量。每当 `g` 被调用时，它都可以访问到 `f` 里的 `a` 变量。即使当 `g` 在 `f` 已经执行完后才被调用，它仍然可以访问及修改 `a`。

```
function f() {
    var a = 1;

    a = 2;
    var b = g();
    a = 3;

    return b;

    function g() {
        return a;
    }
}

f(); // returns 2
```

作用域规则

对于熟悉其它语言的人来说，`var` 声明有些奇怪的作用域规则。看下面的例子：

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }

    return x;
}

f(true);  // returns '10'
f(false); // returns 'undefined'
```

有些读者可能要多看几遍这个例子。 变量 `x` 是定义在 *`if`* 语句里面，但是我们却可以在语句的外面访问它。 这是因为 `var` 声明可以在包含它的函数，模块，命名空间或全局作用域内部任何位置被访问（我们后面会详细介绍），包含它的代码块对此没有什么影响。 有些人称此为 *`var` 作用域或函数作用域*。 函数参数也使用函数作用域。

这些作用域规则可能会引发一些错误。 其中之一就是，多次声明同一个变量并不会报错：

```
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

这里很容易看出一些问题，里层的 `for` 循环会覆盖变量 `i`，因为所有 `i` 都引用相同的函数作用域内的变量。 有经验的开发者们很清楚，这些问题可能在代码审查时漏掉，引发无穷的麻烦。

捕获变量怪异之处

快速的猜一下下面的代码会返回什么：

```
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

介绍一下，`setTimeout` 会在若干毫秒的延时后执行一个函数（等待其它代码执行完毕）。

好吧，看一下结果：

```
10
10
10
10
10
10
10
10
10
10
```

很多JavaScript程序员对这种行为已经很熟悉了，但如果你很不解，你并不是一个人。 大多数人期望输出结果是这样：

```
0
1
2
3
4
5
6
7
8
9
```

还记得我们上面提到的捕获变量吗？ 我们传给 `setTimeout` 的每一个函数表达式实际上都引用了相同作用域里的同一个 `i` 。

让我们花点时间思考一下这是为什么。 `setTimeout` 在若干毫秒后执行一个函数，并且是在 `for` 循环结束后。 `for` 循环结束后， `i` 的值为 `10` 。 所以当函数被调用的时候，它会打印出 `10` ！

一个通常的解决方法是使用立即执行的函数表达式（IIFE）来捕获每次迭代时 `i` 的值：

```
for (var i = 0; i < 10; i++) {
    // capture the current state of 'i'
    // by invoking a function with its current value
    (function(i) {
        setTimeout(function() { console.log(i); }, 100 * i);
    })(i);
}
```

这种奇怪的形式我们已经司空见惯了。 参数 `i` 会覆盖 `for` 循环里的 `i` ，但是因为我们起了同样的名字，所以我们不用怎么改 `for` 循环体里的代码。

`let` 声明

现在你已经知道了 `var` 存在一些问题，这恰好说明了为什么用 `let` 语句来声明变量。 除了名字不同外， `let` 与 `var` 的写法一致。

```
let hello = "Hello!";
```

主要的区别不在语法上，而是语义，我们接下来会深入研究。

块作用域

当用 `let` 声明一个变量，它使用的是词法作用域或块作用域。 不同于使用 `var` 声明的变量那样可以在包含它们的函数外访问，块作用域变量在包含它们的块或 `for` 循环之外是不能访问的。

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        let b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

这里我们定义了2个变量 `a` 和 `b` 。 `a` 的作用域是 `f` 函数体内，而 `b` 的作用域是 `if` 语句块里。

在 `catch` 语句里声明的变量也具有同样的作用域规则。

```
try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

拥有块级作用域的变量的另一个特点是，它们不能在被声明之前读或写。 虽然这些变量始终“存在”于它们的作用域里，但在直到声明它的代码之前的区域都属于暂时性死区。 它只是用来说明我们不能在 `let` 语句之前访问它们，幸运的是TypeScript可以告诉我们这些信息。

```
a++; // illegal to use 'a' before it's declared;
let a;
```

注意一点，我们仍然可以在一个拥有块作用域变量被声明前获取它。 只是我们不能在变量声明前去调用那个函数。 如果生成代码目标为ES2015，现代的运行时会抛出一个错误；然而，现今TypeScript是不会报错的。

```
function foo() {
    // okay to capture 'a'
    return a;
}

// 不能在'a'被声明前调用'foo'
// 运行时应该抛出错误
foo();

let a;
```

关于暂时性死区的更多信息，查看这里Mozilla Developer Network.

重定义及屏蔽

我们提过使用 `var` 声明时，它不在乎你声明多少次；你只会得到1个。

```
function f(x) {
    var x;
    var x;

    if (true) {
        var x;
    }
}
```

在上面的例子里，所有 `x` 的声明实际上都引用一个相同的 `x` ，并且这是完全有效的代码。 这经常会成为bug的来源。 好的是， `let` 声明就不会这么宽松了。

```
let x = 10;
let x = 20; // 错误，不能在1个作用域里多次声明`x`
```

并不是要求两个均是块级作用域的声明TypeScript才会给出一个错误的警告。

```
function f(x) {
    let x = 100; // error: interferes with parameter declaration
}

function g() {
    let x = 100;
    var x = 100; // error: can't have both declarations of 'x'
}
```

并不是说块级作用域变量不能用函数作用域变量来声明。 而是块级作用域变量需要在明显不同的块里声明。

```
function f(condition, x) {
    if (condition) {
        let x = 100;
        return x;
    }

    return x;
}

f(false, 0); // returns 0
f(true, 0);  // returns 100
```

在一个嵌套作用域里引入一个新名字的行为称做屏蔽。 它是一把双刃剑，它可能会不小心地引入新问题，同时也可能会解决一些错误。 例如，假设我们现在用 `let` 重写之前的 `sumMatrix` 函数。

```
function sumMatrix(matrix: number[][]) {
    let sum = 0;
    for (let i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (let i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

这个版本的循环能得到正确的结果，因为内层循环的 `i` 可以屏蔽掉外层循环的 `i` 。

通常来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。同时也有些场景适合利用它，你需要好好打算一下。

块级作用域变量的获取

在我们最初谈及获取用 `var` 声明的变量时，我们简略地探究了一下在获取到了变量之后它的行为是怎样的。直观地讲，每次进入一个作用域时，它创建了一个变量的环境。就算作用域内代码已经执行完毕，这个环境与其捕获的变量依然存在。

```
function theCityThatAlwaysSleeps() {
    let getCity;

    if (true) {
        let city = "Seattle";
        getCity = function() {
            return city;
        }
    }

    return getCity();
}
```

因为我们已经在 `city` 的环境里获取到了 `city` ，所以就算 `if` 语句执行结束后我们仍然可以访问它。

回想一下前面 `setTimeout` 的例子，我们最后需要使用立即执行的函数表达式来获取每次 `for` 循环迭代里的状态。 实际上，我们做的是为获取到的变量创建了一个新的变量环境。 这样做挺痛苦的，但是幸运的是，你不必在TypeScript里这样做了。

当 `let` 声明出现在循环体里时拥有完全不同的行为。 不仅是在循环里引入了一个新的变量环境，而是针对每次迭代都会创建这样一个新作用域。 这就是我们在使用立即执行的函数表达式时做的事，所以在 `setTimeout` 例子里我们仅使用 `let` 声明就可以了。

```
for (let i = 0; i < 10 ; i++) {
    setTimeout(function() {console.log(i); }, 100 * i);
}
```

会输出与预料一致的结果：

```
0
1
2
3
4
5
6
7
8
9
```

`const` 声明

`const` 声明是声明变量的另一种方式。

```
const numLivesForCat = 9;
```

它们与 `let` 声明相似，但是就像它的名字所表达的，它们被赋值后不能再改变。换句话说，它们拥有与 `let` 相同的作用域规则，但是不能对它们重新赋值。

这很好理解，它们引用的值是不可变的。

```
const numLivesForCat = 9;
const kitty = {
    name: "Aurora",
    numLives: numLivesForCat,
}

// Error
kitty = {
    name: "Danielle",
    numLives: numLivesForCat
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;
```

除非你使用特殊的方法去避免，实际上 `const` 变量的内部状态是可修改的。 幸运的是，TypeScript允许你将对象的成员设置成只读的。 接口一章有详细说明。

`let` vs. `const`

现在我们有两种作用域相似的声明方式，我们自然会问到底应该使用哪个。 与大多数泛泛的问题一样，答案是：依情况而定。

使用最小特权原则，所有变量除了你计划去修改的都应该使用 `const` 。 基本原则就是如果一个变量不需要对它写入，那么其它使用这些代码的人也不能够写入它们，并且要思考为什么会需要对这些变量重新赋值。 使用 `const` 也可以让我们更容易的推测数据的流动。

跟据你的自己判断，如果合适的话，与团队成员商议一下。

这个手册大部分地方都使用了 `let` 声明。

解构

Another TypeScript已经可以解析其它 ECMAScript 2015 特性了。完整列表请参见 the article on the Mozilla Developer Network。本章，我们将给出一个简短的概述。

解构数组

最简单的解构莫过于数组的解构赋值了：

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

这创建了2个命名变量 `first` 和 `second` 。相当于使用了索引，但更为方便：

```
first = input[0];
second = input[1];
```

解构作用于已声明的变量会更好：

```
// swap variables
[first, second] = [second, first];
```

作用于函数参数：

```
function f([first, second]: [number, number]) {
    console.log(first);
    console.log(second);
}
f(input);
```

你可以在数组里使用 `...` 语法创建剩余变量：

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

当然，由于是JavaScript, 你可以忽略你不关心的尾随元素：

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

或其它元素：

```
let [, second, , fourth] = [1, 2, 3, 4];
```

对象解构

你也可以解构对象：

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
};
let { a, b } = o;
```

这通过 `o.a` and `o.b` 创建了 `a` 和 `b` 。注意，如果你不需要 `c` 你可以忽略它。

就像数组解构，你可以用没有声明的赋值：

```
({ a, b } = { a: "baz", b: 101 });
```

注意，我们需要用括号将它括起来，因为Javascript通常会将以 `{` 起始的语句解析为一个块。

你可以在对象里使用 `...` 语法创建剩余变量：

```
let { a, ...passthrough } = o;
let total = passthrough.b + passthrough.c.length;
```

属性重命名

你也可以给属性以不同的名字：

```
let { a: newName1, b: newName2 } = o;
```

这里的语法开始变得混乱。 你可以将 `a: newName1` 读做 " `a` 作为 `newName1` "。 方向是从左到右，好像你写成了以下样子：

```
let newName1 = o.a;
let newName2 = o.b;
```

令人困惑的是，这里的冒号不是指示类型的。 如果你想指定它的类型， 仍然需要在其后写上完整的模式。

```
let {a, b}: {a: string, b: number} = o;
```

默认值

默认值可以让你在属性为 undefined 时使用缺省值：

```
function keepWholeObject(wholeObject: { a: string, b?: number })
{
    let { a, b = 1001 } = wholeObject;
}
```

现在，即使 `b` 为 undefined ， `keepWholeObject` 函数的变量 `wholeObject` 的属性 `a` 和 `b` 都会有值。

函数声明

解构也能用于函数声明。 看以下简单的情况：

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
    // ...
}
```

但是，通常情况下更多的是指定默认值，解构默认值有些棘手。 首先，你需要在默认值之前设置其格式。

```
function f({ a, b } = { a: "", b: 0 }): void {
    // ...
}
f(); // ok, default to { a: "", b: 0 }
```

> 上面的代码是一个类型推断的例子，将在本手册后文介绍。

其次，你需要知道在解构属性上给予一个默认或可选的属性用来替换主初始化列表。 要知道 `C` 的定义有一个 `b` 可选属性：

```
function f({ a, b = 0 } = { a: "" }): void {
    // ...
}
f({ a: "yes" }); // ok, default b = 0
f(); // ok, default to {a: ""}, which then defaults b = 0
f({}); // error, 'a' is required if you supply an argument
```

要小心使用解构。 从前面的例子可以看出，就算是最简单的解构表达式也是难以理解的。 尤其当存在深层嵌套解构的时候，就算这时没有堆叠在一起的重命名，默认值和类型注解，也是令人难以理解的。 解构表达式要尽量保持小而简单。 你自己也可以直接使用解构将会生成的赋值表达式。

展开

展开操作符正与解构相反。 它允许你将一个数组展开为另一个数组，或将一个对象展开为另一个对象。 例如：

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

这会令 `bothPlus` 的值为 `[0, 1, 2, 3, 4, 5]` 。 展开操作创建了 `first` 和 `second` 的一份浅拷贝。 它们不会被展开操作所改变。

你还可以展开对象：

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" }
;
let search = { ...defaults, food: "rich" };
```

`search` 的值为 `{ food: "rich", price: "$$", ambiance: "noisy" }` 。 对象的展开比数组的展开要复杂的多。 像数组展开一样，它是从左至右进行处理，但结果仍为对象。 这就意味着出现在展开对象后面的属性会覆盖前面的属性。 因此，如果我们修改上面的例子，在结尾处进行展开的话：

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" }
;
let search = { food: "rich", ...defaults };
```

那么， `defaults` 里的 `food` 属性会重写 `food: "rich"` ，在这里这并不是我们想要的结果。

对象展开还有其它一些意想不到的限制。 首先，它仅包含对象 自身的可枚举属性。 大体上是说当你展开一个对象实例时，你会丢失其方法：

```
class C {
  p = 12;
  m() {
  }
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

其次，TypeScript编译器不允许展开泛型函数上的类型参数。 这个特性会在TypeScript的未来版本中考虑实现。

介绍

TypeScript的核心原则之一是对值所具有的结构进行类型检查。 它有时被称做“鸭式辨型法”或“结构性子类型化”。 在TypeScript里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约。

接口初探

下面通过一个简单示例来观察接口是如何工作的：

```
function printLabel(labelledObj: { label: string }) {
  console.log(labelledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

类型检查器会查看 `printLabel` 的调用。 `printLabel` 有一个参数，并要求这个对象参数有一个名为 `label` 类型为 `string` 的属性。 需要注意的是，我们传入的对象参数实际上会包含很多属性，但是编译器只会检查那些必需的属性是否存在，并且其类型是否匹配。 然而，有些时候TypeScript却并不会这么宽松，我们下面会稍做讲解。

下面我们重写上面的例子，这次使用接口来描述：必须包含一个 `label` 属性且类型为 `string` ：

```
interface LabelledValue {
  label: string;
}

function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

`LabelledValue` 接口就好比一个名字，用来描述上面例子里的要求。 它代表了有一个 `label` 属性且类型为 `string` 的对象。 需要注意的是，我们在这里并不能像在其它语言里一样，说传给 `printLabel` 的对象实现了这个接口。我们只会去关注值的外形。 只要传入的对象满足上面提到的必要条件，那么它就是被允许的。

还有一点值得提的是，类型检查器不会去检查属性的顺序，只要相应的属性存在并且类型也是对的就可以。

可选属性

接口里的属性不全都是必需的。 有些是只在某些条件下存在，或者根本不存在。 可选属性在应用“option bags”模式时很常用，即给函数传入的参数对象中只有部分属性赋值了。

下面是应用了“option bags”的例子：

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; are
a: number} {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个 `?` 符号。

可选属性的好处之一是可以对可能存在的属性进行预定义，好处之二是可以捕获引用了不存在的属性时的错误。 比如，我们故意将 `createSquare` 里的 `color` 属性名拼错，就会得到一个错误提示：

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; ar
ea: number } {
  let newSquare = {color: "white", area: 100};
  if (config.clor) {
    // Error: Property 'clor' does not exist on type 'SquareConf
ig'
    newSquare.color = config.clor;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

只读属性

一些对象属性只能在对象刚刚创建的时候修改其值。你可以在属性名前用 `readonly` 来指定只读属性:

```
interface Point {
    readonly x: number;
    readonly y: number;
}
```

你可以通过赋值一个对象字面量来构造一个 `Point` 。赋值后， `x` 和 `y` 再也不能被改变了。

```
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
```

TypeScript具有 `ReadonlyArray<T>` 类型，它与 `Array<T>` 相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改：

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

上面代码的最后一行，可以看到就算把整个 `ReadonlyArray` 赋值到一个普通数组也是不可以的。 但是你可以用类型断言重写：

```
a = ro as number[];
```

`readonly` vs `const`

最简单判断该用 `readonly` 还是 `const` 的方法是看要把它做为变量使用还是做为一个属性。 做为变量使用的话用 `const` ，若做为属性则使用 `readonly` 。

额外的属性检查

我们在第一个例子里使用了接口，TypeScript让我们传入 `{ size: number; label: string; }` 到仅期望得到 `{ label: string; }` 的函数里。 我们已经学过了可选属性，并且知道他们在“option bags”模式里很有用。

然而，天真地将这两者结合的话就会像在JavaScript里那样搬起石头砸自己的脚。 比如，拿 `createSquare` 例子来说：

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    // ...
}

let mySquare = createSquare({ colour: "red", width: 100 });
```

注意传入 `createSquare` 的参数拼写为 `colour` 而不是 `color` 。 在JavaScript里，这会默默地失败。

你可能会争辩这个程序已经正确地类型化了，因为 `width` 属性是兼容的，不存在 `color` 属性，而且额外的 `colour` 属性是无意义的。

然而，TypeScript会认为这段代码可能存在bug。 对象字面量会被特殊对待而且会经过额外属性检查，当将它们赋值给变量或作为参数传递的时候。 如果一个对象字面量存在任何“目标类型”不包含的属性时，你会得到一个错误。

```
// error: 'colour' not expected in type 'SquareConfig'
let mySquare = createSquare({ colour: "red", width: 100 });
```

绕开这些检查非常简单。 最简便的方法是使用类型断言：

```
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

然而，最佳的方式是能够添加一个字符串索引签名，前提是你能够确定这个对象可能具有某些做为特殊用途使用的额外属性。 如果 `SquareConfig` 带有上面定义的类型的 `color` 和 `width` 属性，并且还会带有任意数量的其它属性，那么我们可以这样定义它：

```
interface SquareConfig {
    color?: string;
    width?: number;
    [propName: string]: any;
}
```

我们稍后会讲到索引签名，但在这我们要表示的是 `SquareConfig` 可以有任意数量的属性，并且只要它们不是 `color` 和 `width` ，那么就无所谓它们的类型是什么。

还有最后一种跳过这些检查的方式，这可能会让你感到惊讶，它就是将这个对象赋值给一个另一个变量： 因为 `squareOptions` 不会经过额外属性检查，所以编译器不会报错。

```
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

要留意，在像上面一样的简单代码里，你可能不应该去绕开这些检查。 对于包含方法和内部状态的复杂对象字面量来讲，你可能需要使用这些技巧，但是大部额外属性检查错误是真正的bug。 就是说你遇到了额外类型检查出的错误，比如“option bags”，你应该去审查一下你的类型声明。 在这里，如果支持传入 `color` 或 `colour` 属性到 `createSquare` ，你应该修改 `SquareConfig` 定义来体现出这一点。

函数类型

接口能够描述JavaScript中对象拥有的各种各样的外形。 除了描述带有属性的普通对象外，接口也可以描述函数类型。

为了使用接口表示函数类型，我们需要给接口定义一个调用签名。 它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

这样定义后，我们可以像使用其它接口一样使用这个函数类型的接口。 下例展示了如何创建一个函数类型的变量，并将一个同类型的函数赋值给这个变量。

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  let result = source.search(subString);
  return result > -1;
}
```

对于函数类型的类型检查来说，函数的参数名不需要与接口里定义的名字相匹配。 比如，我们使用下面的代码重写上面的例子：

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
  let result = src.search(sub);
  return result > -1;
}
```

函数的参数会逐个进行检查，要求对应位置上的参数类型是兼容的。 如果你不想指定类型，TypeScript的类型系统会推断出参数类型，因为函数直接赋值给了 `SearchFunc` 类型变量。 函数的返回值类型是通过其返回值推断出来的（此例是 `false` 和 `true` ）。 如果让这个函数返回数字或字符串，类型检查器会警告我们函数的返回值类型与 `SearchFunc` 接口中的定义不匹配。

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    return result > -1;
}
```

可索引的类型

与使用接口描述函数类型差不多，我们也可以描述那些能够“通过索引得到”的类型，比如 `a[10]` 或 `ageMap["daniel"]` 。 可索引类型具有一个索引签名，它描述了对象索引的类型，还有相应的索引返回值类型。 让我们看一个例子：

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

上面例子里，我们定义了 `StringArray` 接口，它具有索引签名。 这个索引签名表示了当用 `number` 去索引 `StringArray` 时会得到 `string` 类型的返回值。

共有支持两种索引签名：字符串和数字。 可以同时使用两种类型的索引，但是数字索引的返回值必须是字符串索引返回值类型的子类型。 这是因为当使用 `number` 来索引时，JavaScript会将它转换成 `string` 然后再去索引对象。 也就是说用 `100` （一个 `number` ）去索引等同于使用 `"100"` （一个 `string` ）去索引，因此两者需要保持一致。

```
class Animal {
    name: string;
}
class Dog extends Animal {
    breed: string;
}

// 错误：使用数值型的字符串索引，有时会得到完全不同的Animal!
interface NotOkay {
    [x: number]: Animal;
    [x: string]: Dog;
}
```

字符串索引签名能够很好的描述 `dictionary` 模式，并且它们也会确保所有属性与其返回值类型相匹配。 因为字符串索引声明了 `obj.property` 和 `obj["property"]` 两种形式都可以。 下面的例子里， `name` 的类型与字符串索引类型不匹配，所以类型检查器给出一个错误提示：

```
interface NumberDictionary {
  [index: string]: number;
  length: number;    // 可以，length是number类型
  name: string       // 错误，`name`的类型与索引类型返回值的类型不匹配
}
```

最后，你可以将索引签名设置为只读，这样就防止了给索引赋值：

```
interface ReadonlyStringArray {
    readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

你不能设置 `myArray[2]` ，因为索引签名是只读的。

类类型

实现接口

与C#或Java里接口的基本作用一样，TypeScript也能够用它来明确的强制一个类去符合某种契约。

```
interface ClockInterface {
    currentTime: Date;
}

class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

你也可以在接口中描述一个方法，在类里实现它，如同下面的 `setTime` 方法一样：

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

接口描述了类的公共部分，而不是公共和私有两部分。 它不会帮你检查类是否具有某些私有成员。

类静态部分与实例部分的区别

当你操作类和接口的时候，你要知道类是具有两个类型的：静态部分的类型和实例的类型。 你会注意到，当你用构造器签名去定义一个接口并试图定义一个类去实现这个接口时会得到一个错误：

```
interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

这里因为当一个类实现了一个接口时，只对其实例部分进行类型检查。 constructor 存在于类的静态部分，所以不在检查的范围内。

因此，我们应该直接操作类的静态部分。 看下面的例子，我们定义了两个接口， `ClockConstructor` 为构造函数所用和 `ClockInterface` 为实例方法所用。 为了方便我们定义一个构造函数 `createClock` ，它用传入的类型创建实例。

```
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}
interface ClockInterface {
    tick();
}

function createClock(ctor: ClockConstructor, hour: number, minut
e: number): ClockInterface {
    return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("beep beep");
    }
}
class AnalogClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("tick tock");
    }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);
```

因为 `createClock` 的第一个参数是 `ClockConstructor` 类型，在 `createClock(AnalogClock, 7, 32)` 里，会检查 `AnalogClock` 是否符合构造函数签名。

继承接口

和类一样，接口也可以相互继承。 这让我们能够从一个接口里复制成员到另一个接口里，可以更灵活地将接口分割到可重用的模块里。

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

一个接口可以继承多个接口，创建出多个接口的合成接口。

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

混合类型

先前我们提过，接口能够描述JavaScript里丰富的类型。 因为JavaScript其动态灵活的特点，有时你会希望一个对象可以同时具有上面提到的多种类型。

一个例子就是，一个对象可以同时做为函数和对象使用，并带有额外的属性。

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

在使用JavaScript第三方库的时候，你可能需要像上面那样去完整地定义类型。

接口继承类

当接口继承了一个类类型时，它会继承类的成员但不包括其实现。 就好像接口声明了所有类中存在的成员，但并没有提供具体实现一样。 接口同样会继承到类的private和protected成员。 这意味着当你创建了一个接口继承了一个拥有私有或受保护的成员的类时，这个接口类型只能被这个类或其子类所实现（implement）。

当你有一个庞大的继承结构时这很有用，但要指出的是你的代码只在子类拥有特定属性时起作用。 除了继承自基类，子类之间不必相关联。 例：

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control implements SelectableControl {
    select() { }
}

class TextBox extends Control {
    select() { }
}

// Error: Property 'state' is missing in type 'Image'.
class Image implements SelectableControl {
    select() { }
}

class Location {

}
```

在上面的例子里，`SelectableControl` 包含了 `Control` 的所有成员，包括私有成员 `state` 。因为 `state` 是私有成员，所以只能够是 `Control` 的子类们才能实现 `SelectableControl` 接口。因为只有 `Control` 的子类才能够拥有一个声明于 `Control` 的私有成员 `state` ，这对私有成员的兼容性是必需的。

在 `Control` 类内部，是允许通过 `SelectableControl` 的实例来访问私有成员 `state` 的。实际上，`SelectableControl` 就像 `Control` 一样，并拥有一个 `select` 方法。`Button` 和 `TextBox` 类是 `SelectableControl` 的子类（因为它们都继承自 `Control` 并有 `select` 方法），但 `Image` 和 `Location` 类并不是这样的。

介绍

传统的JavaScript程序使用函数和基于原型的继承来创建可重用的组件，但对于熟悉使用面向对象方式的程序员来讲就有些棘手，因为他们用的是基于类的继承并且对象是由类构建出来的。 从ECMAScript 2015，也就是ECMAScript 6开始，JavaScript程序员将能够使用基于类的面向对象的方式。 使用TypeScript，我们允许开发者现在就使用这些特性，并且编译后的JavaScript可以在所有主流浏览器和平台上运行，而不需要等到下个JavaScript版本。

类

下面看一个使用类的例子：

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```

如果你使用过C#或Java，你会对这种语法非常熟悉。 我们声明一个 `Greeter` 类。这个类有3个成员：一个叫做 `greeting` 的属性，一个构造函数和一个 `greet` 方法。

你会注意到，我们在引用任何一个类成员的时候都用了 `this` 。 它表示我们访问的是类的成员。

最后一行，我们使用 `new` 构造了 `Greeter` 类的一个实例。 它会调用之前定义的构造函数，创建一个 `Greeter` 类型的新对象，并执行构造函数初始化它。

继承

在TypeScript里，我们可以使用常用的面向对象模式。 基于类的程序设计中一种最基本的模式是允许使用继承来扩展现有的类。

看下面的例子：

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

这个例子展示了最基本的继承：类从基类中继承了属性和方法。 这里， `Dog` 是一个派生类，它派生自 `Animal` 基类，通过 `extends` 关键字。 派生类通常被称作子类，基类通常被称作超类。

因为 `Dog` 继承了 `Animal` 的功能，因此我们可以创建一个 `Dog` 的实例，它能够 `bark()` 和 `move()` 。

下面我们来看个更加复杂的例子。

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

这个例子展示了一些上面没有提到的特性。 这一次，我们使用 `extends` 关键字创建了 `Animal` 的两个子类： `Horse` 和 `Snake` 。

与前一个例子的不同点是，派生类包含了一个构造函数，它必须调用 `super()` ，它会执行基类的构造函数。 而且，在构造函数里访问 `this` 的属性之前，我们一定要调用 `super()` 。 这个是TypeScript强制执行的一条重要规则。

这个例子演示了如何在子类里可以重写父类的方法。 `Snake` 类和 `Horse` 类都创建了 `move` 方法，它们重写了从 `Animal` 继承来的 `move` 方法，使得 `move` 方法根据不同的类而具有不同的功能。 注意，即使 `tom` 被声明为 `Animal` 类型，但因为它的值是 `Horse` ，调用 `tom.move(34)` 时，它会调用 `Horse` 里重写的方法：

```
Slithering...
Sammy the Python moved 5m.
Galloping...
Tommy the Palomino moved 34m.
```

公共，私有与受保护的修饰符

默认为 `public`

在上面的例子里，我们可以自由的访问程序里定义的成员。 如果你对其它语言中的类比较了解，就会注意到我们在之前的代码里并没有使用 `public` 来做修饰；例如，C#要求必须明确地使用 `public` 指定成员是可见的。 在TypeScript里，成员都默认为 `public` 。

你也可以明确的将一个成员标记成 `public` 。 我们可以用下面的方式来重写上面的 `Animal` 类：

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

理解 `private`

当成员被标记成 `private` 时，它就不能在声明它的类的外部访问。比如：

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // 错误: 'name' 是私有的.
```

TypeScript使用的是结构性类型系统。 当我们比较两种不同的类型时，并不在乎它们从何处而来，如果所有成员的类型都是兼容的，我们就认为它们的类型是兼容的。

然而，当我们比较带有 `private` 或 `protected` 成员的类型的时候，情况就不同了。 如果其中一个类型里包含一个 `private` 成员，那么只有当另外一个类型中也存在这样一个 `private` 成员， 并且它们都是来自同一处声明时，我们才认为这两个类型是兼容的。 对于 `protected` 成员也使用这个规则。

下面来看一个例子，更好地说明了这一点：

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
    constructor() { super("Rhino"); }
}

class Employee {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // 错误: Animal 与 Employee 不兼容.
```

这个例子中有 `Animal` 和 `Rhino` 两个类， `Rhino` 是 `Animal` 类的子类。 还有一个 `Employee` 类，其类型看上去与 `Animal` 是相同的。 我们创建了几个这些类的实例，并相互赋值来看看会发生什么。 因为 `Animal` 和 `Rhino` 共享了来自 `Animal` 里的私有成员定义 `private name: string` ，因此它们是兼容的。 然而 `Employee` 却不是这样。当把 `Employee` 赋值给 `Animal` 的时候，得到一个错误，说它们的类型不兼容。 尽管 `Employee` 里也有一个私有成员 `name` ，但它明显不是 `Animal` 里面定义的那个。

理解 `protected`

`protected` 修饰符与 `private` 修饰符的行为很相似，但有一点不同， `protected` 成员在派生类中仍然可以访问。例如：

```
class Person {
    protected name: string;
    constructor(name: string) { this.name = name; }
}

class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name)
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // 错误
```

注意，我们不能在 `Person` 类外使用 `name`，但是我们仍然可以通过 `Employee` 类的实例方法访问，因为 `Employee` 是由 `Person` 派生而来的。

构造函数也可以被标记成 `protected`。 这意味着这个类不能在包含它的类外被实例化，但是能被继承。比如，

```
class Person {
    protected name: string;
    protected constructor(theName: string) { this.name = theName
; }
}

// Employee 能够继承 Person
class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${t
his.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // 错误: 'Person' 的构造函数是被保护
的.
```

readonly修饰符

你可以使用 `readonly` 关键字将属性设置为只读的。 只读属性必须在声明时或构造函数里被初始化。

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // 错误! name 是只读的.
```

参数属性

在上面的例子中，我们不得不定义一个受保护的成员 `name` 和一个构造函数参数 `theName` 在 `Person` 类里，并且立刻将 `theName` 的值赋给 `name` 。 这种情况经常会遇到。参数属性可以方便地让我们在一个地方定义并初始化一个成员。 下面的例子是对之前 `Animal` 类的修改版，使用了参数属性：

```
class Animal {
    constructor(private name: string) { }
    move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

注意看我们是如何舍弃了 `theName` ，仅在构造函数里使用 `private name: string` 参数来创建和初始化 `name` 成员。 我们把声明和赋值合并至一处。

参数属性通过给构造函数参数添加一个访问限定符来声明。 使用 `private` 限定一个参数属性会声明并初始化一个私有成员；对于 `public` 和 `protected` 来说也是一样。

存取器

TypeScript支持通过getters/setters来截取对对象成员的访问。 它能帮助你有效的控制对对象成员的访问。

下面来看如何把一个简单的类改写成使用 `get` 和 `set` 。 首先，我们从一个没有使用存取器的例子开始。

```
class Employee {
    fullName: string;
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

我们可以随意的设置 `fullName` ，这是非常方便的，但是这也可能会带来麻烦。

下面这个版本里，我们先检查用户密码是否正确，然后再允许其修改员工信息。 我们把对 `fullName` 的直接访问改成了可以检查密码的 `set` 方法。 我们也加了一个 `get` 方法，让上面的例子仍然可以工作。

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!"
);
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

我们可以修改一下密码，来验证一下存取器是否是工作的。当密码不对时，会提示我们没有权限去修改员工。

对于存取器有下面几点需要注意的：

首先，存取器要求你将编译器设置为输出ECMAScript 5或更高。 不支持降级到ECMAScript 3。 其次，只带有 `get` 不带有 `set` 的存取器自动被推断为 `readonly` 。 这在从代码生成 `.d.ts` 文件时是有帮助的，因为利用这个属性的用户会看到不允许够改变它的值。

静态属性

到目前为止，我们只讨论了类的实例成员，那些仅当类被实例化的时候才会被初始化的属性。 我们也可以创建类的静态成员，这些属性存在于类本身上面而不是类的实例上。 在这个例子里，我们使用 `static` 定义 `origin` ，因为它是所有网格都会用到的属性。 每个实例想要访问这个属性的时候，都要在 `origin` 前面加上类名。 如同在实例属性上使用 `this.` 前缀来访问属性一样，这里我们使用 `Grid.` 来访问静态属性。

```
class Grid {
    static origin = {x: 0, y: 0};
    calculateDistanceFromOrigin(point: {x: number; y: number;})
{
        let xDist = (point.x - Grid.origin.x);
        let yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.s
cale;
    }
    constructor (public scale: number) { }
}

let grid1 = new Grid(1.0);  // 1x scale
let grid2 = new Grid(5.0);  // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

抽象类

抽象类做为其它派生类的基类使用。 它们一般不会直接被实例化。 不同于接口，抽象类可以包含成员的实现细节。 `abstract` 关键字是用于定义抽象类和在抽象类内部定义抽象方法。

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log('roaming the earch...');
    }
}
```

抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。 抽象方法的语法与接口方法相似。 两者都是定义方法签名但不包含方法体。 然而，抽象方法必须包含 `abstract` 关键字并且可以包含访问修饰符。

```
abstract class Department {

    constructor(public name: string) {
    }

    printName(): void {
        console.log('Department name: ' + this.name);
    }

    abstract printMeeting(): void; // 必须在派生类中实现
}

class AccountingDepartment extends Department {

    constructor() {
        super('Accounting and Auditing'); // 在派生类的构造函数中必须调用 super()
    }

    printMeeting(): void {
        console.log('The Accounting Department meets each Monday at 10am.');
    }

    generateReports(): void {
        console.log('Generating accounting reports...');
    }
}

let department: Department; // 允许创建一个对抽象类型的引用
department = new Department(); // 错误: 不能创建一个抽象类的实例
department = new AccountingDepartment(); // 允许对一个抽象子类进行实例化和赋值
department.printName();
department.printMeeting();
department.generateReports(); // 错误: 方法在声明的抽象类中不存在
```

高级技巧

构造函数

当你在TypeScript里声明了一个类的时候，实际上同时声明了很多东西。 首先就是类的实例的类型。

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

这里，我们写了 `let greeter: Greeter` ，意思是 `Greeter` 类的实例的类型是 `Greeter` 。 这对于用过其它面向对象语言的程序员来讲已经是老习惯了。

我们也创建了一个叫做构造函数的值。 这个函数会在我们使用 `new` 创建类实例的时候被调用。 下面我们来看看，上面的代码被编译成JavaScript后是什么样子的：

```
let Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

上面的代码里， `let Greeter` 将被赋值为构造函数。 当我们调用 `new` 并执行了这个函数后，便会得到一个类的实例。 这个构造函数也包含了类的所有静态属性。 换个角度说，我们可以认为类具有实例部分与静态部分这两个部分。

让我们稍微改写一下这个例子，看看它们之间的区别：

```
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        }
        else {
            return Greeter.standardGreeting;
        }
    }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());
```

这个例子里， `greeter1` 与之前看到的一样。 我们实例化 `Greeter` 类，并使用这个对象。 与我们之前看到的一样。

再之后，我们直接使用类。 我们创建了一个叫做 `greeterMaker` 的变量。 这个变量保存了这个类或者说保存了类构造函数。 然后我们使用 `typeof Greeter` ，意思是取Greeter类的类型，而不是实例的类型。 或者更确切的说，"告诉我 `Greeter` 标识符的类型"，也就是构造函数的类型。 这个类型包含了类的所有静态成员和构造函数。 之后，就和前面一样，我们在 `greeterMaker` 上使用 `new` ，创建 `Greeter` 的实例。

把类当做接口使用

如上一节里所讲的，类定义会创建两个东西：类的实例类型和一个构造函数。 因为类可以创建出类型，所以你能够在允许使用接口的地方使用类。

```
class Point {
    x: number;
    y: number;
}

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

介绍

函数是JavaScript应用程序的基础。 它帮助你实现抽象层，模拟类，信息隐藏和模块。 在TypeScript里，虽然已经支持类，命名空间和模块，但函数仍然是主要的定义行为的地方。 TypeScript为JavaScript函数添加了额外的功能，让我们可以更容易地使用。

函数

和JavaScript一样，TypeScript函数可以创建有名字的函数和匿名函数。 你可以随意选择适合应用程序的方式，不论是定义一系列API函数还是只使用一次的函数。

通过下面的例子可以迅速回想起这两种JavaScript中的函数：

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

在JavaScript里，函数可以使用函数体外部的变量。 当函数这么做时，我们说它'捕获'了这些变量。 至于为什么可以这样做以及其中的利弊超出了本文的范围，但是深刻理解这个机制对学习JavaScript和TypeScript会很有帮助。

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

函数类型

为函数定义类型

让我们为上面那个函数添加类型：

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x +
y; };
```

我们可以给每个参数添加类型之后再为函数本身添加返回值类型。 TypeScript能够根据返回语句自动推断出返回值类型，因此我们通常省略它。

书写完整函数类型

现在我们已经为函数指定了类型，下面让我们写出函数的完整类型。

```
let myAdd: (x:number, y:number) => number =
    function(x: number, y: number): number { return x + y; };
```

函数类型包含两部分：参数类型和返回值类型。 当写出完整函数类型的时候，这两部分都是需要的。 我们以参数列表的形式写出参数类型，为每个参数指定一个名字和类型。 这个名字只是为了增加可读性。 我们也可以这么写：

```
let myAdd: (baseValue: number, increment: number) => number =
    function(x: number, y: number): number { return x + y; };
```

只要参数类型是匹配的，那么就认为它是有效的函数类型，而不在乎参数名是否正确。

第二部分是返回值类型。 对于返回值，我们在函数和返回值类型之前使用(`=>`)符号，使之清晰明了。 如之前提到的，返回值类型是函数类型的必要部分，如果函数没有返回任何值，你也必须指定返回值类型为 `void` 而不能留空。

函数的类型只是由参数类型和返回值组成的。 函数中使用的捕获变量不会体现在类型里。 实际上，这些变量是函数的隐藏状态并不是组成API的一部分。

推断类型

尝试这个例子的时候，你会发现如果你在赋值语句的一边指定了类型但是另一边没有类型的话，TypeScript编译器会自动识别出类型：

```
// myAdd has the full function type
let myAdd = function(x: number, y: number): number { return x +
y; };

// The parameters `x` and `y` have the type number
let myAdd: (baseValue: number, increment: number) => number =
    function(x, y) { return x + y; };
```

这叫做“按上下文归类”，是类型推论的一种。 它帮助我们更好地为程序指定类型。

可选参数和默认参数

TypeScript里的每个函数参数都是必须的。 这不是指不能传递 `null` 或 `undefined` 作为参数，而是说编译器检查用户是否为每个参数都传入了值。 编译器还会假设只有这些参数会被传递进函数。 简短地说，传递给一个函数的参数个数必须与函数期望的参数个数一致。

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too f
ew parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too m
any parameters
let result3 = buildName("Bob", "Adams");         // ah, just rig
ht
```

JavaScript里，每个参数都是可选的，可传可不传。 没传参的时候，它的值就是undefined。 在TypeScript里我们可以在参数名旁使用 `?` 实现可选参数的功能。 比如，我们想让last name是可选的：

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob");  // works correctly now
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too m
any parameters
let result3 = buildName("Bob", "Adams");  // ah, just right
```

可选参数必须跟在必须参数后面。 如果上例我们想让first name是可选的，那么就必须调整它们的位置，把first name放在后面。

在TypeScript里，我们也可以为参数提供一个默认值当用户没有传递这个参数或传递的值是 `undefined` 时。 它们叫做有默认初始化值的参数。 让我们修改上例，把last name的默认值设置为 `"Smith"` 。

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // works correc
tly now, returns "Bob Smith"
let result2 = buildName("Bob", undefined);       // still works,
 also returns "Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr.");  // error, too m
any parameters
let result4 = buildName("Bob", "Adams");         // ah, just rig
ht
```

在所有必须参数后面的带默认初始化的参数都是可选的，与可选参数一样，在调用函数的时候可以省略。 也就是说可选参数与末尾的默认参数共享参数类型。

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

和

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
```

共享同样的类型 `(firstName: string, lastName?: string) => string` 。 默认参数的默认值消失了，只保留了它是一个可选参数的信息。

与普通可选参数不同的是，带默认值的参数不需要放在必须参数的后面。 如果带默认值的参数出现在必须参数前面，用户必须明确的传入 `undefined` 值来获得默认值。 例如，我们重写最后一个例子，让 `firstName` 是带默认值的参数：

```
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too f
ew parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too m
any parameters
let result3 = buildName("Bob", "Adams");         // okay and ret
urns "Bob Adams"
let result4 = buildName(undefined, "Adams");     // okay and ret
urns "Will Adams"
```

剩余参数

必要参数，默认参数和可选参数有个共同点：它们表示某一个参数。 有时，你想同时操作多个参数，或者你并不知道会有多少参数传递进来。 在JavaScript里，你可以使用 `arguments` 来访问所有传入的参数。

在TypeScript里，你可以把所有参数收集到一个变量里：

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKi
nzie");
```

剩余参数会被当做个数不限的可选参数。 可以一个都没有，同样也可以有任意个。 编译器创建参数数组，名字是你在省略号（ `...` ）后面给定的名字，你可以在函数体内使用这个数组。

这个省略号也会在带有剩余参数的函数类型定义上使用到：

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string =
 buildName;
```

`this`

学习如何在JavaScript里正确使用 `this` 就好比一场成年礼。 由于TypeScript是JavaScript的超集，TypeScript程序员也需要弄清 `this` 工作机制并且当有bug的时候能够找出错误所在。 幸运的是，TypeScript能通知你错误地使用了 `this` 的地方。 如果你想了解JavaScript里的 `this` 是如何工作的，那么首先阅读Yehuda Katz写的Understanding JavaScript Function Invocation and "this"。 Yehuda的文章详细的阐述了 `this` 的内部工作原理，因此我们这里只做简单介绍。

`this` 和箭头函数

JavaScript里， `this` 的值在函数被调用的时候才会指定。 这是个既强大又灵活的特点，但是你需要花点时间弄清楚函数调用的上下文是什么。 但众所周知，这不是一件很简单的事，尤其是在返回一个函数或将函数当做参数传递的时候。

下面看一个例子：

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        return function() {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCa
rd % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

可以看到 `createCardPicker` 是个函数，并且它又返回了一个函数。 如果我们尝试运行这个程序，会发现它并没有弹出对话框而是报错了。 因为 `createCardPicker` 返回的函数里的 `this` 被设置成了 `window` 而不是 `deck` 对象。 因为我们只是独立的调用了 `cardPicker()` 。 顶级的非方法式调用会将 `this` 视为 `window` 。 （注意：在严格模式下， `this` 为 `undefined` 而不是 `window` ）。

为了解决这个问题，我们可以在函数被返回时就绑好正确的 `this` 。 这样的话，无论之后怎么使用它，都会引用绑定的‘deck’对象。 我们需要改变函数表达式来使用ECMAScript 6箭头语法。 箭头函数能保存函数创建时的 `this` 值，而不是调用时的值：

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // NOTE: the line below is now an arrow function, allowi
ng us to capture 'this' right here
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCa
rd % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

更好事情是，TypeScript会警告你犯了一个错误，如果你给编译器设置了 `--noImplicitThis` 标记。它会指出 `this.suits[pickedSuit]` 里的 `this` 的类型为 `any` 。

`this` 参数

不幸的是， `this.suits[pickedSuit]` 的类型依旧为 `any` 。这是因为 `this` 来自对象字面量里的函数表达式。修改的方法是，提供一个显式的 `this` 参数。 `this` 参数是个假的参数，它出现在参数列表的最前面：

```
function f(this: void) {
    // make sure `this` is unusable in this standalone function
}
```

让我们往例子里添加一些接口， `Card` 和 `Deck` ，让类型重用能够变得清晰简单些：

```
interface Card {
    suit: string;
    card: number;
}
interface Deck {
    suits: string[];
    cards: number[];
    createCardPicker(this: Deck): () => Card;
}
let deck: Deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    // NOTE: The function now explicitly specifies that its call
ee must be of type Deck
    createCardPicker: function(this: Deck) {
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCa
rd % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

现在TypeScript知道 `createCardPicker` 期望在某个 `Deck` 对象上调用。也就是说 `this` 是 `Deck` 类型的，而非 `any` ，因此 `--noImplicitThis` 不会报错了。

回调函数里的 `this` 参数

当你将一个函数传递到某个库函数里在稍后被调用时，你可能也见到过回调函数里的 `this` 会报错。因为当回调函数被调用时，它会被当成一个普通函数调用， `this` 将为 `undefined` 。稍做改动，你就可以通过 `this` 参数来避免错

误。 首先，库函数的作者要指定 `this` 的类型：

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` 意味着 `addClickListener` 期望 `onclick` 是一个函数且它不需要一个 `this` 类型。 然后，为调用代码里的 `this` 添加类型注解：

```
class Handler {
    info: string;
    onClickBad(this: Handler, e: Event) {
        // oops, used this here. using this callback would crash at runtime
        this.info = e.message;
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // error!
```

指定了 `this` 类型后，你显式声明 `onClickBad` 必须在 `Handler` 的实例上调用。然后TypeScript会检测到 `addClickListener` 要求函数带有 `this: void` 。 改变 `this` 类型来修复这个错误：

```
class Handler {
    info: string;
    onClickGood(this: void, e: Event) {
        // can't use this here because it's of type void!
        console.log('clicked!');
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickGood);
```

因为 `onClickGood` 指定了 `this` 类型为 `void` ，因此传递 `addClickListener` 是合法的。 当然了，这也意味着不能使用 `this.info` . 如果你两者都想要，你不得不使用箭头函数了：

```
class Handler {
    info: string;
    onClickGood = (e: Event) => { this.info = e.message }
}
```

这是可行的因为箭头函数不会捕获 `this` ，所以你总是可以把它们传给期望 `this: void` 的函数。 缺点是每个 `Handler` 对象都会创建一个箭头函数。 另一方面，方法只会被创建一次，添加到 `Handler` 的原型链上。 它们在不同 `Handler` 对象间是共享的。

重载

JavaScript本身是个动态语言。 JavaScript里函数根据传入不同的参数而返回不同类型的数据是很常见的。

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", c
ard: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

`pickCard` 方法根据传入参数的不同会返回两种不同的类型。 如果传入的是代表纸牌的对象，函数作用是从中抓一张牌。 如果用户想抓牌，我们告诉他抓到了什么牌。 但是这怎么在类型系统里表示呢。

方法是为同一个函数提供多个函数类型定义来进行函数重载。 编译器会根据这个列表去处理函数的调用。 下面我们来重载 `pickCard` 函数。

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", c
ard: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

这样改变后，重载的 `pickCard` 函数在调用的时候会进行正确的类型检查。

为了让编译器能够选择正确的检查类型，它与JavaScript里的处理流程相似。 它查找重载列表，尝试使用第一个重载定义。 如果匹配的话就使用这个。 因此，在定义重载的时候，一定要把最精确的定义放在最前面。

注意， `function pickCard(x): any` 并不是重载列表的一部分，因此这里只有两个重载：一个是接收对象另一个接收数字。 以其它参数调用 `pickCard` 会产生错误。

介绍

软件工程中，我们不仅要创建一致的定义良好的API，同时也要考虑可重用性。 组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

在像C#和Java这样的语言中，可以使用 `泛型` 来创建可重用的组件，一个组件可以支持多种类型的数据。 这样用户就可以以自己的数据类型来使用组件。

泛型之Hello World

下面来创建第一个使用泛型的例子：identity函数。 这个函数会返回任何传入它的值。 你可以把这个函数当成是 `echo` 命令。

不用泛型的话，这个函数可能是下面这样：

```
function identity(arg: number): number {
    return arg;
}
```

或者，我们使用 `any` 类型来定义函数：

```
function identity(arg: any): any {
    return arg;
}
```

使用 `any` 类型会导致这个函数可以接收任何类型的 `arg` 参数，这样就丢失了一些信息：传入的类型与返回的类型应该是相同的。 如果我们传入一个数字，我们只知道任何类型的值都有可能被返回。

因此，我们需要一种方法使返回值的类型与传入参数的类型是相同的。 这里，我们使用了类型变量，它是一种特殊的变量，只用于表示类型而不是值。

```
function identity<T>(arg: T): T {
    return arg;
}
```

我们给identity添加了类型变量 `T` 。 `T` 帮助我们捕获用户传入的类型（比如： `number` ），之后我们就可以使用这个类型。 之后我们再次使用了 `T` 当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。 这允许我们跟踪函数里使用的类型的信息。

我们把这个版本的 `identity` 函数叫做泛型，因为它可以适用于多个类型。 不同于使用 `any` ，它不会丢失信息，像第一个例子那像保持准确性，传入数值类型并返回数值类型。

我们定义了泛型函数后，可以用两种方法使用。 第一种是，传入所有的参数，包含类型参数：

```
let output = identity<string>("myString");  // type of output wi
ll be 'string'
```

这里我们明确的指定了 `T` 是 `string` 类型，并做为一个参数传给函数，使用了 `<>` 括起来而不是 `()` 。

第二种方法更普遍。利用了类型推论 -- 即编译器会根据传入的参数自动地帮助我们确定T的类型：

```
let output = identity("myString");  // type of output will be 's
tring'
```

注意我们没必要使用尖括号（ `<>` ）来明确地传入类型；编译器可以查看 `myString` 的值，然后把 `T` 设置为它的类型。 类型推论帮助我们保持代码精简和高可读性。如果编译器不能够自动地推断出类型的话，只能像上面那样明确的传入T的类型，在一些复杂的情况下，这是可能出现的。

使用泛型变量

使用泛型创建像 `identity` 这样的泛型函数时，编译器要求你在函数体必须正确的使用这个通用的类型。 换句话说，你必须把这些参数当做是任意或所有类型。

看下之前 `identity` 例子：

```
function identity<T>(arg: T): T {
    return arg;
}
```

如果我们想同时打印出 `arg` 的长度。 我们很可能会这样做：

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

如果这么做，编译器会报错说我们使用了 `arg` 的 `.length` 属性，但是没有地方指明 `arg` 具有这个属性。 记住，这些类型变量代表的是任意类型，所以使用这个函数的人可能传入的是个数字，而数字是没有 `.length` 属性的。

现在假设我们想操作 `T` 类型的数组而不直接是 `T` 。由于我们操作的是数组，所以 `.length` 属性是应该存在的。 我们可以像创建其它数组一样创建这个数组：

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array has a .length, so no more
 error
    return arg;
}
```

你可以这样理解 `loggingIdentity` 的类型：泛型函数 `loggingIdentity` ，接收类型参数 `T` 和参数 `arg` ，它是个元素类型是 `T` 的数组，并返回元素类型是 `T` 的数组。 如果我们传入数字数组，将返回一个数字数组，因为此时 `T` 的的类型为 `number` 。 这可以让我们把泛型变量T当做类型的一部分使用，而不是整个类型，增加了灵活性。

我们也可以这样实现上面的例子：

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array has a .length, so no more
 error
    return arg;
}
```

使用过其它语言的话，你可能对这种语法已经很熟悉了。 在下一节，会介绍如何创建自定义泛型像 `Array<T>` 一样。

泛型类型

上一节，我们创建了identity通用函数，可以适用于不同的类型。 在这节，我们研究一下函数本身的类型，以及如何创建泛型接口。

泛型函数的类型与非泛型函数的类型没什么不同，只是有一个类型参数在最前面，像函数声明一样：

```
function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: <T>(arg: T) => T = identity;
```

我们也可以使用不同的泛型参数名，只要在数量上和使用方式上能对应上就可以。

```
function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: <U>(arg: U) => U = identity;
```

我们还可以使用带有调用签名的对象字面量来定义泛型函数：

```
function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: {<T>(arg: T): T} = identity;
```

这引导我们去写第一个泛型接口了。 我们把上面例子里的对象字面量拿出来做为一个接口：

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: GenericIdentityFn = identity;
```

一个相似的例子，我们可能想把泛型参数当作整个接口的一个参数。 这样我们就能清楚的知道使用的具体是哪个泛型类型（比如： `Dictionary<string>而不只是 Dictionary` ）。 这样接口里的其它成员也能知道这个参数的类型了。

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;
```

注意，我们的示例做了少许改动。 不再描述泛型函数，而是把非泛型函数签名作为泛型类型一部分。 当我们使用 `GenericIdentityFn` 的时候，还得传入一个类型参数来指定泛型类型（这里是： `number` ），锁定了之后代码里使用的类型。 对

于描述哪部分类型属于泛型部分来说，理解何时把参数放在调用签名里和何时放在接口上是很有帮助的。

除了泛型接口，我们还可以创建泛型类。 注意，无法创建泛型枚举和泛型命名空间。

泛型类

泛型类看上去与泛型接口差不多。 泛型类使用（ `<>` ）括起泛型类型，跟在类名后面。

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

`GenericNumber` 类的使用是十分直观的，并且你可能已经注意到了，没有什么去限制它只能使用 `number` 类型。 也可以使用字符串或其它更复杂的类型。

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

与接口一样，直接把泛型类型放在类后面，可以帮助我们确认类的所有属性都在使用相同的类型。

我们在类那节说过，类有两部分：静态部分和实例部分。 泛型类指的是实例部分的类型，所以类的静态属性不能使用这个泛型类型。

泛型约束

你应该会记得之前的一个例子，我们有时候想操作某类型的一组值，并且我们知道这组值具有什么样的属性。 在 `loggingIdentity` 例子中，我们想访问 `arg` 的 `length` 属性，但是编译器并不能证明每种类型都有 `length` 属性，所以就报错了。

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

相比于操作any所有类型，我们想要限制函数去处理任意带有 `.length` 属性的所有类型。 只要传入的类型有这个属性，我们就允许，就是说至少包含这一属性。 为此，我们需要列出对于T的约束要求。

为此，我们定义一个接口来描述约束条件。 创建一个包含 `.length` 属性的接口，使用这个接口和 `extends` 关键字来实现约束：

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no more error
    return arg;
}
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3);  // Error, number doesn't have a .length property
```

我们需要传入符合约束类型的值，必须包含必须的属性：

```
loggingIdentity({length: 10, value: 3});
```

在泛型约束中使用类型参数

你可以声明一个类型参数，且它被另一个类型参数所约束。 比如，现在我们想要用属性名从对象里获取这个属性。 并且我们想要确保这个属性存在于对象 `obj` 上，因此我们需要在这两个类型之间使用约束。

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a"); // okay
getProperty(x, "m"); // error: Argument of type 'm' isn't assign
able to 'a' | 'b' | 'c' | 'd'.
```

在泛型里使用类类型

在TypeScript使用泛型创建工厂函数时，需要引用构造函数的类类型。比如，

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

一个更高级的例子，使用原型属性推断并约束构造函数与类实例的关系。

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag;  // typechecks!
createInstance(Bee).keeper.hasMask;   // typechecks!
```

枚举

使用枚举我们可以定义一些带名字的常量。 使用枚举可以清晰地表达意图或创建一组有区别的用例。 TypeScript支持数字的和基于字符串的枚举。

数字枚举

首先我们看看数字枚举，如果你使用过其它编程语言应该会很熟悉。

```
enum Direction {
    Up = 1,
    Down,
    Left,
    Right
}
```

如上，我们定义了一个数字枚举， `Up` 使用初始化为 `1` 。 其余的成员会从 `1` 开始自动增长。 换句话说， `Direction.Up` 的值为 `1` ， `Down` 为 `2` ， `Left` 为 `3` ， `Right` 为 `4` 。

我们还可以完全不使用初始化器：

```
enum Direction {
    Up,
    Down,
    Left,
    Right,
}
```

现在， `Up` 的值为 `0` ， `Down` 的值为 `1` 等等。 当我们不在乎成员的值的时候，这种自增长的行为是很有用处的，但是要注意每个枚举成员的值都是不同的。

使用枚举很简单：通过枚举的属性来访问枚举成员，和枚举的名字来访问枚举类型：

```
enum Response {
    No = 0,
    Yes = 1,
}

function respond(recipient: string, message: Response): void {
    // ...
}

respond("Princess Caroline", Response.Yes)
```

数字枚举可以被混入到计算过的和常量成员（如下所示）。简短地说，不带初始化器的枚举或者被放在第一的位置，或者被放在使用了数字常量或其它常量初始化了的枚举后面。换句话说，下面的情况是不被允许的：

```
enum E {
    A = getSomeValue(),
    B, // error! 'A' is not constant-initialized, so 'B' needs a
n initializer
}
```

字符串枚举

字符串枚举的概念很简单，但是有细微的运行时的差别。在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化。

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

由于字符串枚举没有自增长的行为，字符串枚举可以很好的序列化。换句话说，如果你正在调试并且必须要读一个数字枚举的运行时的值，这个值通常是很难读的 - 它并不能表达有用的信息（尽管反向映射会有所帮助），字符串枚举允许你提供一

个运行时有意义的并且可读的值，独立于枚举成员的名字。

异构枚举（Heterogeneous enums）

从技术的角度来说，枚举可以混合字符串和数字成员，但是似乎你并不会这么做：

```
enum BooleanLikeHeterogeneousEnum {
    No = 0,
    Yes = "YES",
}
```

除非你真的想要利用JavaScript运行时的行为，否则我们不建议这样做。

计算的和常量成员

每个枚举成员都带有一个值，它可以是常量或计算出来的。 当满足如下条件时，枚举成员被当作是常量：

- 它是枚举的第一个成员且没有初始化器，这种情况下它被赋予值 `0`：

  ```
  // E.X is constant:
  enum E { X }
  ```

- 它不带有初始化器且它之前的枚举成员是一个数字常量。 这种情况下，当前枚举成员的值为它上一个枚举成员的值加1。

  ```
  // All enum members in 'E1' and 'E2' are constant.

  enum E1 { X, Y, Z }

  enum E2 {
      A = 1, B, C
  }
  ```

- 枚举成员使用常量枚举表达式初始化。 常数枚举表达式是TypeScript表达式的子集，它可以在编译阶段求值。 当一个表达式满足下面条件之一时，它就是一个常量枚举表达式：
  - 一个枚举表达式字面量（主要是字符串字面量或数字字面量）
  - 一个对之前定义的常量枚举成员的引用（可以是在不同的枚举类型中定义的）
  - 带括号的常量枚举表达式
  - 一元运算符 `+`, `-`, `~` 其中之一应用在了常量枚举表达式
  - 常量枚举表达式做为二元运算符 `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, `^` 的操作对象。 若常数枚举表达式求值后为 `NaN` 或 `Infinity`，则会在编译阶段报错。

所有其它情况的枚举成员被当作是需要计算得出的值。

```
enum FileAccess {
    // constant members
    None,
    Read    = 1 << 1,
    Write   = 1 << 2,
    ReadWrite  = Read | Write,
    // computed member
    G = "123".length
}
```

联合枚举与枚举成员的类型

存在一种特殊的非计算的常量枚举成员的子集：字面量枚举成员。 字面量枚举成员是指不带有初始值的常量枚举成员，或者是值被初始化为

- 任何字符串字面量（例如： `"foo"`， `"bar"`， `"baz"`）
- 任何数字字面量（例如： `1`, `100`）
- 应用了一元 `-` 符号的数字字面量（例如： `-1`, `-100`）

当所有枚举成员都拥有字面量枚举值时，它就带有了一种特殊的语义。

首先，枚举成员成为了类型！ 例如，我们可以说某些成员只能是枚举成员的值：

```
enum ShapeKind {
    Circle,
    Square,
}

interface Circle {
    kind: ShapeKind.Circle;
    radius: number;
}

interface Square {
    kind: ShapeKind.Square;
    sideLength: number;
}

let c: Circle = {
    kind: ShapeKind.Square,
    //    ~~~~~~~~~~~~~~~~ Error!
    radius: 100,
}
```

另一个变化是枚举类型本身变成了每个枚举成员的联合。 虽然我们还没有讨论联合类型，但你只要知道通过联合枚举，类型系统能够利用这样一个事实，它可以知道枚举里的值的集合。 因此，TypeScript能够捕获在比较值的时候犯的愚蠢的错误。 例如：

```
enum E {
    Foo,
    Bar,
}

function f(x: E) {
    if (x !== E.Foo || x !== E.Bar) {
        //             ~~~~~~~~~~~
        // Error! Operator '!==' cannot be applied to types 'E.Foo' and 'E.Bar'.
    }
}
```

这个例子里，我们先检查 `x` 是否不是 `E.Foo` 。 如果通过了这个检查，然后 `||` 会发生短路效果， `if` 语句体里的内容会被执行。 然而，这个检查没有通过，那么 `x` 则只能为 `E.Foo` ，因此没理由再去检查它是否为 `E.Bar` 。

运行时的枚举

枚举是在运行时真正存在的对象。 例如下面的枚举：

```
enum E {
    X, Y, Z
}
```

can actually be passed around to functions

```
function f(obj: { X: number }) {
    return obj.X;
}

// Works, since 'E' has a property named 'X' which is a number.
f(E);
```

反向映射

除了创建一个以属性名做为对象成员的对象之外，数字枚举成员还具有了反向映射，从枚举值到枚举名字。 例如，在下面的例子中：

```
enum Enum {
    A
}
let a = Enum.A;
let nameOfA = Enum[a]; // "A"
```

TypeScript可能会将这段代码编译为下面的JavaScript：

```
var Enum;
(function (Enum) {
    Enum[Enum["A"] = 0] = "A";
})(Enum || (Enum = {}));
var a = Enum.A;
var nameOfA = Enum[a]; // "A"
```

生成的代码中，枚举类型被编译成一个对象，它包含了正向映射（`name` -> `value`）和反向映射（`value` -> `name`）。引用枚举成员总会生成为对属性访问并且永远也不会内联代码。

要注意的是不会为字符串枚举成员生成反向映射。

`const` 枚举

大多数情况下，枚举是十分有效的方案。然而在某些情况下需求很严格。为了避免在额外生成的代码上的开销和额外的非直接的对枚举成员的访问，我们可以使用 `const` 枚举。常量枚举通过在枚举上使用 `const` 修饰符来定义。

```
const enum Enum {
    A = 1,
    B = A * 2
}
```

常量枚举只能使用常量枚举表达式，并且不同于常规的枚举，它们在编译阶段会被删除。常量枚举成员在使用的地方会被内联进来。之所以可以这么做是因为，常量枚举不允许包含计算成员。

```
const enum Directions {
    Up,
    Down,
    Left,
    Right
}

let directions = [Directions.Up, Directions.Down, Directions.Lef
t, Directions.Right]
```

生成后的代码为：

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* R
ight */];
```

外部枚举

外部枚举用来描述已经存在的枚举类型的形状。

```
declare enum Enum {
    A = 1,
    B,
    C = 2
}
```

外部枚举和非外部枚举之间有一个重要的区别，在正常的枚举里，没有初始化方法的成员被当成常数成员。 对于非常数的外部枚举而言，没有初始化方法时被当做需要经过计算的。

介绍

这节介绍TypeScript里的类型推论。即，类型是在哪里如何被推断的。

基础

TypeScript里，在有些没有明确指出类型的地方，类型推论会帮助提供类型。如下面的例子

```
let x = 3;
```

变量 `x` 的类型被推断为数字。 这种推断发生在初始化变量和成员，设置默认参数值和决定函数返回值时。

大多数情况下，类型推论是直截了当地。 后面的小节，我们会浏览类型推论时的细微差别。

最佳通用类型

当需要从几个表达式中推断类型时候，会使用这些表达式的类型来推断出一个最合适的通用类型。例如，

```
let x = [0, 1, null];
```

为了推断 `x` 的类型，我们必须考虑所有元素的类型。 这里有两种选择： `number` 和 `null` 。 计算通用类型算法会考虑所有的候选类型，并给出一个兼容所有候选类型的类型。

由于最终的通用类型取自候选类型，有些时候候选类型共享相同的通用类型，但是却没有一个类型能做为所有候选类型的类型。例如：

```
let zoo = [new Rhino(), new Elephant(), new Snake()];
```

这里，我们想让zoo被推断为 `Animal[]` 类型，但是这个数组里没有对象是 `Animal` 类型的，因此不能推断出这个结果。 为了更正，当候选类型不能使用的时候我们需要明确的指出类型：

```
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

如果没有找到最佳通用类型的话，类型推断的结果为联合数组类型， `(Rhino | Elephant | Snake)[]` 。

上下文类型

TypeScript类型推论也可能按照相反的方向进行。 这被叫做“按上下文归类”。按上下文归类会发生在表达式的类型与所处的位置相关时。比如：

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.button);  //<- Error
};
```

这个例子会得到一个类型错误，TypeScript类型检查器使用 `Window.onmousedown` 函数的类型来推断右边函数表达式的类型。 因此，就能推断出 `mouseEvent` 参数的类型了。 如果函数表达式不是在上下文类型的位置， `mouseEvent` 参数的类型需要指定为 `any` ，这样也不会报错了。

如果上下文类型表达式包含了明确的类型信息，上下文的类型被忽略。 重写上面的例子：

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.button);  //<- Now, no error is given
};
```

这个函数表达式有明确的参数类型注解，上下文类型被忽略。 这样的话就不报错了，因为这里不会使用到上下文类型。

上下文归类会在很多情况下使用到。 通常包含函数的参数，赋值表达式的右边，类型断言，对象成员和数组字面量和返回值语句。 上下文类型也会做为最佳通用类型的候选类型。比如：

```
function createZoo(): Animal[] {
    return [new Rhino(), new Elephant(), new Snake()];
}
```

这个例子里，最佳通用类型有4个候选者：`Animal`，`Rhino`，`Elephant`和`Snake`。当然，`Animal`会被做为最佳通用类型。

介绍

TypeScript里的类型兼容性是基于结构子类型的。 结构类型是一种只使用其成员来描述类型的方式。 它正好与名义（nominal）类型形成对比。（译者注：在基于名义类型的类型系统中，数据类型的兼容性或等价性是通过明确的声明和/或类型的名称来决定的。这与结构性类型系统不同，它是基于类型的组成结构，且不要求明确地声明。） 看下面的例子：

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

在使用基于名义类型的语言，比如C#或Java中，这段代码会报错，因为Person类没有明确说明其实现了Named接口。

TypeScript的结构性子类型是根据JavaScript代码的典型写法来设计的。 因为JavaScript里广泛地使用匿名对象，例如函数表达式和对象字面量，所以使用结构类型系统来描述这些类型比使用名义类型系统更好。

关于可靠性的注意事项

TypeScript的类型系统允许某些在编译阶段无法确认其安全性的操作。当一个类型系统具此属性时，被当做是“不可靠”的。TypeScript允许这种不可靠行为的发生是经过仔细考虑的。通过这篇文章，我们会解释什么时候会发生这种情况和其有利的一面。

开始

TypeScript结构化类型系统的基本规则是，如果 `x` 要兼容 `y` ，那么 `y` 至少具有与 `x` 相同的属性。比如：

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: 'Alice', location: 'Seattle' };
x = y;
```

这里要检查 `y` 是否能赋值给 `x` ，编译器检查 `x` 中的每个属性，看是否能在 `y` 中也找到对应属性。 在这个例子中， `y` 必须包含名字是 `name` 的 `string` 类型成员。 `y` 满足条件，因此赋值正确。

检查函数参数时使用相同的规则：

```
function greet(n: Named) {
    alert('Hello, ' + n.name);
}
greet(y); // OK
```

注意， `y` 有个额外的 `location` 属性，但这不会引发错误。 只有目标类型（这里是 `Named` ）的成员会被一一检查是否兼容。

这个比较过程是递归进行的，检查每个成员及子成员。

比较两个函数

相对来讲，在比较原始类型和对象类型的时候是比较容易理解的，问题是如何判断两个函数是兼容的。 下面我们从两个简单的函数入手，它们仅是参数列表略有不同：

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

要查看 `x` 是否能赋值给 `y` ，首先看它们的参数列表。 `x` 的每个参数必须能在 `y` 里找到对应类型的参数。 注意的是参数的名字相同与否无所谓，只看它们的类型。 这里， `x` 的每个参数在 `y` 中都能找到对应的参数，所以允许赋值。

第二个赋值错误，因为 `y` 有个必需的第二个参数，但是 `x` 并没有，所以不允许赋值。

你可能会疑惑为什么允许 `忽略` 参数，像例子 `y = x` 中那样。 原因是忽略额外的参数在JavaScript里是很常见的。 例如， `Array#forEach` 给回调函数传3个参数：数组元素，索引和整个数组。 尽管如此，传入一个只使用第一个参数的回调函数也是很有用的：

```
let items = [1, 2, 3];

// Don't force these extra arguments
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach((item) => console.log(item));
```

下面来看看如何处理返回值类型，创建两个仅是返回值类型不同的函数：

```
let x = () => ({name: 'Alice'});
let y = () => ({name: 'Alice', location: 'Seattle'});

x = y; // OK
y = x; // Error because x() lacks a location property
```

类型系统强制源函数的返回值类型必须是目标函数返回值类型的子类型。

函数参数双向协变

当比较函数参数类型时，只有当源函数参数能够赋值给目标函数或者反过来时才能赋值成功。 这是不稳定的，因为调用者可能传入了一个具有更精确类型信息的函数，但是调用这个传入的函数的时候却使用了不是那么精确的类型信息。 实际上，这极少会发生错误，并且能够实现很多JavaScript里的常见模式。例如：

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) =
> void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x
+ ',' + e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEv
ent>e).x + ',' + (<MouseEvent>e).y));
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent
) => console.log(e.x + ',' + e.y)));

// Still disallowed (clear error). Type safety enforced for whol
ly incompatible types
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

可选参数及剩余参数

比较函数兼容性的时候，可选参数与必须参数是可互换的。 源类型上有额外的可选参数不是错误，目标类型的可选参数在源类型里没有对应的参数也不是错误。

当一个函数有剩余参数时，它被当做无限个可选参数。

这对于类型系统来说是不稳定的，但从运行时的角度来看，可选参数一般来说是不强制的，因为对于大多数函数来说相当于传递了一些 `undefinded` 。

有一个好的例子，常见的函数接收一个回调函数并用对于程序员来说是可预知的参数但对类型系统来说是不确定的参数来调用：

```
function invokeLater(args: any[], callback: (...args: any[]) =>
void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ', ' + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ', ' + y));
```

函数重载

对于有重载的函数，源函数的每个重载都要在目标函数上找到对应的函数签名。 这确保了目标函数可以在所有源函数可调用的地方调用。

枚举

枚举类型与数字类型兼容，并且数字类型与枚举类型兼容。不同枚举类型之间是不兼容的。比如，

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

let status = Status.Ready;
status = Color.Green;  //error
```

类

类与对象字面量和接口差不多，但有一点不同：类有静态部分和实例部分的类型。比较两个类类型的对象时，只有实例的成员会被比较。 静态成员和构造函数不在比较的范围内。

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s;  //OK
s = a;  //OK
```

类的私有成员和受保护成员

类的私有成员和受保护成员会影响兼容性。 当检查类实例的兼容时，如果目标类型包含一个私有成员，那么源类型必须包含来自同一个类的这个私有成员。 同样地，这条规则也适用于包含受保护成员实例的类型检查。 这允许子类赋值给父类，但是不能赋值给其它有同样类型的类。

泛型

因为TypeScript是结构性的类型系统，类型参数只影响使用其做为类型一部分的结果类型。比如，

```
interface Empty<T> {
}
let x: Empty<number>;
let y: Empty<string>;

x = y;  // okay, y matches structure of x
```

上面代码里，`x` 和 `y` 是兼容的，因为它们的结构使用类型参数时并没有什么不同。 把这个例子改变一下，增加一个成员，就能看出是如何工作的了：

```
interface NotEmpty<T> {
    data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y;  // error, x and y are not compatible
```

在这里，泛型类型在使用时就好比不是一个泛型类型。

对于没指定泛型类型的泛型参数时，会把所有泛型参数当成 `any` 比较。 然后用结果类型进行比较，就像上面第一个例子。

比如，

```
let identity = function<T>(x: T): T {
    // ...
}

let reverse = function<U>(y: U): U {
    // ...
}

identity = reverse;  // Okay because (x: any)=>any matches (y: any)=>any
```

高级主题

子类型与赋值

目前为止，我们使用了“兼容性”，它在语言规范里没有定义。 在TypeScript里，有两种兼容性：子类型和赋值。 它们的不同点在于，赋值扩展了子类型兼容性，增加了一些规则，允许和 `any` 来回赋值，以及 `enum` 和对应数字值之间的来回赋值。

语言里的不同地方分别使用了它们之中的机制。 实际上，类型兼容性是由赋值兼容性来控制的，即使在 `implements` 和 `extends` 语句也不例外。

更多信息，请参阅TypeScript语言规范.

交叉类型（Intersection Types）

交叉类型是将多个类型合并为一个类型。 这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。 例如， `Person & Serializable & Loggable` 同时是 `Person` 和 `Serializable` 和 `Loggable` 。 就是说这个类型的对象同时拥有了这三种类型的成员。

我们大多是在混入（mixins）或其它不适合典型面向对象模型的地方看到交叉类型的使用。 （在JavaScript里发生这种情况的场合很多！） 下面是如何创建混入的一个简单例子("target": "es5")：

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

联合类型（Union Types）

联合类型与交叉类型很有关联，但是使用上却完全不同。 偶尔你会遇到这种情况，一个代码库希望传入 `number` 或 `string` 类型的参数。 例如下面的函数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: any) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

`padLeft` 存在一个问题， `padding` 参数的类型指定成了 `any` 。 这就是说我们可以传入一个既不是 `number` 也不是 `string` 类型的参数，但是TypeScript却不报错。

```
let indentedString = padLeft("Hello world", true); // 编译阶段通过，运行时报错
```

在传统的面向对象语言里，我们可能会将这两种类型抽象成有层级的类型。 这么做显然是非常清晰的，但同时也存在了过度设计。 `padLeft` 原始版本的好处之一是允许我们传入原始类型。 这样做的话使用起来既简单又方便。 如果我们就是想使用已经存在的函数的话，这种新的方式就不适用了。

代替 `any` ， 我们可以使用联合类型做为 `padding` 的参数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

联合类型表示一个值可以是几种类型之一。 我们用竖线（ `|` ）分隔每个类型，所以 `number | string | boolean` 表示一个值可以是 `number` ， `string` ，或 `boolean` 。

如果一个值是联合类型，我们只能访问此联合类型的所有类型里共有的成员。

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();    // errors
```

这里的联合类型可能有点复杂，但是你很容易就习惯了。 如果一个值的类型是 `A | B`，我们能够确定的是它包含了 `A` 和 `B` 中共有的成员。 这个例子里， `Bird` 具有一个 `fly` 成员。 我们不能确定一个 `Bird | Fish` 类型的变量是否有 `fly` 方法。 如果变量在运行时是 `Fish` 类型，那么调用 `pet.fly()` 就出错了。

类型保护与区分类型（Type Guards and Differentiating Types）

联合类型适合于那些值可以为不同类型的情况。 但当我们想确切地了解是否为 `Fish` 时怎么办？ JavaScript里常用来区分2个可能值的方法是检查成员是否存在。 如之前提及的，我们只能访问联合类型中共同拥有的成员。

```
let pet = getSmallPet();

// 每一个成员访问都会报错
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}
```

为了让这段代码工作，我们要使用类型断言：

```
let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}
```

用户自定义的类型保护

这里可以注意到我们不得不多次使用类型断言。 假若我们一旦检查过类型，就能在之后的每个分支里清楚地知道 `pet` 的类型的话就好了。

TypeScript里的类型保护机制让它成为了现实。 类型保护就是一些表达式，它们会在运行时检查以确保在某个作用域里的类型。 要定义一个类型保护，我们只要简单地定义一个函数，它的返回值是一个类型谓词：

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}
```

在这个例子里， `pet is Fish` 就是类型谓词。 谓词为 `parameterName is Type` 这种形式， `parameterName` 必须是来自于当前函数签名里的一个参数名。

每当使用一些变量调用 `isFish` 时，TypeScript会将变量缩减为那个具体的类型，只要这个类型与变量的原始类型是兼容的。

```
// 'swim' 和 'fly' 调用都没有问题了

if (isFish(pet)) {
    pet.swim();
}
else {
    pet.fly();
}
```

注意TypeScript不仅知道在 `if` 分支里 `pet` 是 `Fish` 类型； 它还清楚在 `else` 分支里，一定不是 `Fish` 类型，一定是 `Bird` 类型。

`typeof` 类型保护

现在我们回过头来看看怎么使用联合类型书写 `padLeft` 代码。 我们可以像下面这样利用类型断言来写：

```
function isNumber(x: any): x is number {
    return typeof x === "number";
}

function isString(x: any): x is string {
    return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
    if (isNumber(padding)) {
        return Array(padding + 1).join(" ") + value;
    }
    if (isString(padding)) {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'
.`);
}
```

然而，必须要定义一个函数来判断类型是否是原始类型，这太痛苦了。 幸运的是，现在我们不必将 `typeof x === "number"` 抽象成一个函数，因为TypeScript可以将它识别为一个类型保护。 也就是说我们可以直接在代码里检查类型了。

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'
.`);
}
```

这些 *`typeof`* 类型保护只有两种形式能被识别： `typeof v === "typename"` 和 `typeof v !== "typename"` ， `"typename"` 必须是 `"number"` ， `"string"` ， `"boolean"` 或 `"symbol"` 。 但是TypeScript并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

`instanceof` 类型保护

如果你已经阅读了 `typeof` 类型保护并且对JavaScript里的 `instanceof` 操作符熟悉的话，你可能已经猜到了这节要讲的内容。

`instanceof` 类型保护是通过构造函数来细化类型的一种方式。 比如，我们借鉴一下之前字符串填充的例子：

```
interface Padder {
    getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
    constructor(private numSpaces: number) { }
    getPaddingString() {
        return Array(this.numSpaces + 1).join(" ");
    }
}

class StringPadder implements Padder {
    constructor(private value: string) { }
    getPaddingString() {
        return this.value;
    }
}

function getRandomPadder() {
    return Math.random() < 0.5 ?
        new SpaceRepeatingPadder(4) :
        new StringPadder("  ");
}

// 类型为SpaceRepeatingPadder | StringPadder
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
    padder; // 类型细化为'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
    padder; // 类型细化为'StringPadder'
}
```

`instanceof` 的右侧要求是一个构造函数，TypeScript将细化为：

1. 此构造函数的 `prototype` 属性的类型，如果它的类型不为 `any` 的话
2. 构造签名所返回的类型的联合

以此顺序。

可以为null的类型

TypeScript具有两种特殊的类型， `null` 和 `undefined` ，它们分别具有值null和undefined. 我们在基础类型一节里已经做过简要说明。 默认情况下，类型检查器认为 `null` 与 `undefined` 可以赋值给任何类型。 `null` 与 `undefined` 是所有其它类型的一个有效值。 这也意味着，你阻止不了将它们赋值给其它类型，就算是你想要阻止这种情况也不行。 `null` 的发明者，Tony Hoare，称它为价值亿万美金的错误。

`--strictNullChecks` 标记可以解决此错误：当你声明一个变量时，它不会自动地包含 `null` 或 `undefined` 。 你可以使用联合类型明确的包含它们：

```
let s = "foo";
s = null; // 错误, 'null'不能赋值给'string'
let sn: string | null = "bar";
sn = null; // 可以

sn = undefined; // error, 'undefined'不能赋值给'string | null'
```

注意，按照JavaScript的语义，TypeScript会把 `null` 和 `undefined` 区别对待。 `string | null` ， `string | undefined` 和 `string | undefined | null` 是不同的类型。

可选参数和可选属性

使用了 `--strictNullChecks` ，可选参数会被自动地加上 `| undefined` :

```
function f(x: number, y?: number) {
    return x + (y || 0);
}
f(1, 2);
f(1);
f(1, undefined);
f(1, null); // error, 'null' is not assignable to 'number | unde
fined'
```

可选属性也会有同样的处理：

```
class C {
    a: number;
    b?: number;
}
let c = new C();
c.a = 12;
c.a = undefined; // error, 'undefined' is not assignable to 'num
ber'
c.b = 13;
c.b = undefined; // ok
c.b = null; // error, 'null' is not assignable to 'number | unde
fined'
```

类型保护和类型断言

由于可以为null的类型是通过联合类型实现，那么你需要使用类型保护来去除 `null` 。幸运地是这与在JavaScript里写的代码一致：

```
function f(sn: string | null): string {
    if (sn == null) {
        return "default";
    }
    else {
        return sn;
    }
}
```

这里很明显地去除了 `null` ，你也可以使用短路运算符：

```
function f(sn: string | null): string {
    return sn || "default";
}
```

如果编译器不能够去除 `null` 或 `undefined` ，你可以使用类型断言手动去除。语法是添加 `!` 后缀： `identifier!` 从 `identifier` 的类型里去除了 `null` 和 `undefined` ：

```
function broken(name: string | null): string {
  function postfix(epithet: string) {
    return name.charAt(0) + '.  the ' + epithet; // error, 'name
' is possibly null
  }
  name = name || "Bob";
  return postfix("great");
}

function fixed(name: string | null): string {
  function postfix(epithet: string) {
    return name!.charAt(0) + '.  the ' + epithet; // ok
  }
  name = name || "Bob";
  return postfix("great");
}
```

本例使用了嵌套函数，因为编译器无法去除嵌套函数的null（除非是立即调用的函数表达式）。因为它无法跟踪所有对嵌套函数的调用，尤其是你将内层函数做为外层函数的返回值。如果无法知道函数在哪里被调用，就无法知道调用时 `name` 的类型。

类型别名

类型别名会给一个类型起个新名字。类型别名有时和接口很像，但是可以作用于原始值，联合类型，元组以及其它任何你需要手写的类型。

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    }
    else {
        return n();
    }
}
```

起别名不会新建一个类型 - 它创建了一个新名字来引用那个类型。给原始类型起别名通常没什么用，尽管可以做为文档的一种形式使用。

同接口一样，类型别名也可以是泛型 - 我们可以添加类型参数并且在别名声明的右侧传入：

```
type Container<T> = { value: T };
```

我们也可以使用类型别名来在属性里引用自己：

```
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

与交叉类型一起使用，我们可以创建出一些十分稀奇古怪的类型。

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
```

然而，类型别名不能出现在声明右侧的任何地方。

```
type Yikes = Array<Yikes>; // error
```

接口 vs. 类型别名

像我们提到的，类型别名可以像接口一样；然而，仍有一些细微差别。

其一，接口创建了一个新的名字，可以在其它任何地方使用。 类型别名并不创建新名字—比如，错误信息就不会使用别名。 在下面的示例代码里，在编译器中将鼠标悬停在 `interfaced` 上，显示它返回的是 `Interface` ，但悬停在 `aliased` 上时，显示的却是对象字面量类型。

```
type Alias = { num: number }
interface Interface {
    num: number;
}
declare function aliased(arg: Alias): Alias;
declare function interfaced(arg: Interface): Interface;
```

另一个重要区别是类型别名不能被 `extends` 和 `implements` （自己也不能 `extends` 和 `implements` 其它类型）。 因为软件中的对象应该对于扩展是开放的，但是对于修改是封闭的，你应该尽量去使用接口代替类型别名。

另一方面，如果你无法通过接口来描述一个类型并且需要使用联合类型或元组类型，这时通常会使用类型别名。

字符串字面量类型

字符串字面量类型允许你指定字符串必须的固定值。 在实际应用中，字符串字面量类型可以与联合类型，类型保护和类型别名很好的配合。 通过结合使用这些特性，你可以实现类似枚举类型的字符串。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowe
d here
```

你只能从三种允许的字符中选择其一来做为参数传递，传入其它值则会产生错误。

```
Argument of type '"uneasy"' is not assignable to parameter of ty
pe '"ease-in" | "ease-out" | "ease-in-out"'
```

字符串字面量类型还可以用于区分函数重载：

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
    // ... code goes here ...
}
```

数字字面量类型

TypeScript还具有数字字面量类型。

```
function rollDie(): 1 | 2 | 3 | 4 | 5 | 6 {
    // ...
}
```

我们很少直接这样使用，但它们可以用在缩小范围调试bug的时候：

```
function foo(x: number) {
    if (x !== 1 || x !== 2) {
        //         ~~~~~~~
        // Operator '!==' cannot be applied to types '1' and '2'.

    }
}
```

换句话说，当 `x` 与 `2` 进行比较的时候，它的值必须为 `1` ，这就意味着上面的比较检查是非法的。

枚举成员类型

如我们在枚举一节里提到的，当每个枚举成员都是用字面量初始化的时候枚举成员是具有类型的。

在我们谈及“单例类型”的时候，多数是指枚举成员类型和数字/字符串字面量类型，尽管大多数用户会互换使用“单例类型”和“字面量类型”。

可辨识联合（Discriminated Unions）

你可以合并单例类型，联合类型，类型保护和类型别名来创建一个叫做可辨识联合的高级模式，它也称做标签联合或代数数据类型。 可辨识联合在函数式编程很有用处。 一些语言会自动地为你辨识联合；而TypeScript则基于已有的JavaScript模式。 它具有3个要素：

1. 具有普通的单例类型属性—可辨识的特征。

2. 一个类型别名包含了那些类型的联合——联合。
3. 此属性上的类型保护。

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}
```

首先我们声明了将要联合的接口。每个接口都有 `kind` 属性但有不同的字符串字面量类型。 `kind` 属性称做可辨识的特征或标签。其它的属性则特定于各个接口。注意，目前各个接口间是没有联系的。下面我们把它们联合到一起：

```
type Shape = Square | Rectangle | Circle;
```

现在我们使用可辨识联合:

```
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

完整性检查

当没有涵盖所有可辨识联合的变化时，我们想让编译器可以通知我们。 比如，如果我们添加了 `Triangle` 到 `Shape` ，我们同时还需要更新 `area` :

```
type Shape = Square | Rectangle | Circle | Triangle;
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
    // should error here - we didn't handle case "triangle"
}
```

有两种方式可以实现。 首先是启用 `--strictNullChecks` 并且指定一个返回值类型：

```
function area(s: Shape): number { // error: returns number | undefined
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

因为 `switch` 没有包涵所有情况，所以TypeScript认为这个函数有时候会返回 `undefined` 。 如果你明确地指定了返回值类型为 `number` ，那么你会看到一个错误，因为实际上返回值的类型为 `number | undefined` 。 然而，这种方法存在些微妙之处且 `--strictNullChecks` 对旧代码支持不好。

第二种方法使用 `never` 类型，编译器用它来进行完整性检查：

```
function assertNever(x: never): never {
    throw new Error("Unexpected object: " + x);
}
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
        default: return assertNever(s); // error here if there a
re missing cases
    }
}
```

这里， `assertNever` 检查 `s` 是否为 `never` 类型—即为除去所有可能情况后剩下的类型。 如果你忘记了某个case，那么 `s` 将具有一个真实的类型并且你会得到一个错误。 这种方式需要你定义一个额外的函数，但是在你忘记某个case的时候也更加明显。

多态的 `this` 类型

多态的 `this` 类型表示的是某个包含类或接口的子类型。 这被称做*F*-bounded多态性。 它能很容易的表现连贯接口间的继承，比如。 在计算器的例子里，在每个操作之后都返回 `this` 类型：

```
class BasicCalculator {
    public constructor(protected value: number = 0) { }
    public currentValue(): number {
        return this.value;
    }
    public add(operand: number): this {
        this.value += operand;
        return this;
    }
    public multiply(operand: number): this {
        this.value *= operand;
        return this;
    }
    // ... other operations go here ...
}

let v = new BasicCalculator(2)
            .multiply(5)
            .add(1)
            .currentValue();
```

由于这个类使用了 `this` 类型，你可以继承它，新的类可以直接使用之前的方法，不需要做任何的改变。

```
class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }
    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
    // ... other operations go here ...
}

let v = new ScientificCalculator(2)
        .multiply(5)
        .sin()
        .add(1)
        .currentValue();
```

如果没有 `this` 类型， `ScientificCalculator` 就不能够在继承 `BasicCalculator` 的同时还保持接口的连贯性。 `multiply` 将会返回 `BasicCalculator` ，它并没有 `sin` 方法。 然而，使用 `this` 类型， `multiply` 会返回 `this` ，在这里就是 `ScientificCalculator` 。

索引类型（Index types）

使用索引类型，编译器就能够检查使用了动态属性名的代码。 例如，一个常见的JavaScript模式是从对象中选取属性的子集。

```
function pluck(o, names) {
    return names.map(n => o[n]);
}
```

下面是如何在TypeScript里使用此函数，通过索引类型查询和索引访问操作符：

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
  return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

编译器会检查 `name` 是否真的是 `Person` 的一个属性。 本例还引入了几个新的类型操作符。 首先是 `keyof T` ，索引类型查询操作符。 对于任何类型 `T` ， `keyof T` 的结果为 `T` 上已知的公共属性名的联合。 例如：

```
let personProps: keyof Person; // 'name' | 'age'
```

`keyof Person` 是完全可以与 `'name' | 'age'` 互相替换的。 不同的是如果你添加了其它的属性到 `Person` ，例如 `address: string` ，那么 `keyof Person` 会自动变为 `'name' | 'age' | 'address'` 。 你可以在像 `pluck` 函数这类上下文里使用 `keyof` ，因为在使用之前你并不清楚可能出现的属性名。 但编译器会检查你是否传入了正确的属性名给 `pluck` ：

```
pluck(person, ['age', 'unknown']); // error, 'unknown' is not in
 'name' | 'age'
```

第二个操作符是 `T[K]` ，索引访问操作符。 在这里，类型语法反映了表达式语法。 这意味着 `person['name']` 具有类型 `Person['name']` — 在我们的例子里则为 `string` 类型。 然而，就像索引类型查询一样，你可以在普通的上下文里使用 `T[K]` ，这正是它的强大所在。 你只要确保类型变量 `K extends keyof T` 就可以了。 例如下面 `getProperty` 函数的例子：

```
function getProperty<T, K extends keyof T>(o: T, name: K): T[K]
{
    return o[name]; // o[name] is of type T[K]
}
```

`getProperty` 里的 `o: T` 和 `name: K` ，意味着 `o[name]: T[K]` 。当你返回 `T[K]` 的结果，编译器会实例化键的真实类型，因此 `getProperty` 的返回值类型会随着你需要的属性改变。

```
let name: string = getProperty(person, 'name');
let age: number = getProperty(person, 'age');
let unknown = getProperty(person, 'unknown'); // error, 'unknown
' is not in 'name' | 'age'
```

索引类型和字符串索引签名

`keyof` 和 `T[K]` 与字符串索引签名进行交互。如果你有一个带有字符串索引签名的类型，那么 `keyof T` 会是 `string` 。并且 `T[string]` 为索引签名的类型：

```
interface Map<T> {
    [key: string]: T;
}
let keys: keyof Map<number>; // string
let value: Map<number>['foo']; // number
```

映射类型

一个常见的任务是将一个已知的类型每个属性都变为可选的：

```
interface PersonPartial {
    name?: string;
    age?: number;
}
```

或者我们想要一个只读版本：

```
interface PersonReadonly {
    readonly name: string;
    readonly age: number;
}
```

这在JavaScript里经常出现，TypeScript提供了从旧类型中创建新类型的一种方式 — 映射类型。 在映射类型里，新类型以相同的形式去转换旧类型里每个属性。 例如，你可以令每个属性成为 `readonly` 类型或可选的。 下面是一些例子：

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
}
type Partial<T> = {
    [P in keyof T]?: T[P];
}
```

像下面这样使用：

```
type PersonPartial = Partial<Person>;
type ReadonlyPerson = Readonly<Person>;
```

下面来看看最简单的映射类型和它的组成部分：

```
type Keys = 'option1' | 'option2';
type Flags = { [K in Keys]: boolean };
```

它的语法与索引签名的语法类型，内部使用了 `for .. in` 。 具有三个部分：

1. 类型变量 `K` ，它会依次绑定到每个属性。
2. 字符串字面量联合的 `Keys` ，它包含了要迭代的属性名的集合。
3. 属性的结果类型。

在个简单的例子里， `Keys` 是硬编码的的属性名列表并且属性类型永远是 `boolean` ，因此这个映射类型等同于：

```
type Flags = {
    option1: boolean;
    option2: boolean;
}
```

在真正的应用里，可能不同于上面的 `Readonly` 或 `Partial` 。 它们会基于一些已存在的类型，且按照一定的方式转换字段。 这就是 `keyof` 和索引访问类型要做的事情：

```
type NullablePerson = { [P in keyof Person]: Person[P] | null }
type PartialPerson = { [P in keyof Person]?: Person[P] }
```

但它更有用的地方是可以有一些通用版本。

```
type Nullable<T> = { [P in keyof T]: T[P] | null }
type Partial<T> = { [P in keyof T]?: T[P] }
```

在这些例子里，属性列表是 `keyof T` 且结果类型是 `T[P]` 的变体。 这是使用通用映射类型的一个好模版。 因为这类转换是同态的，映射只作用于 `T` 的属性而没有其它的。 编译器知道在添加任何新属性之前可以拷贝所有存在的属性修饰符。 例如，假设 `Person.name` 是只读的，那么 `Partial<Person>.name` 也将是只读的且为可选的。

下面是另一个例子， `T[P]` 被包装在 `Proxy<T>` 类里：

```
type Proxy<T> = {
    get(): T;
    set(value: T): void;
}
type Proxify<T> = {
    [P in keyof T]: Proxy<T[P]>;
}
function proxify<T>(o: T): Proxify<T> {
   // ... wrap proxies ...
}
let proxyProps = proxify(props);
```

注意 `Readonly<T>` 和 `Partial<T>` 用处不小，因此它们与 `Pick` 和 `Record` 一同被包含进了TypeScript的标准库里：

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
}
type Record<K extends string, T> = {
    [P in K]: T;
}
```

`Readonly`， `Partial` 和 `Pick` 是同态的，但 `Record` 不是。 因为 `Record` 并不需要输入类型来拷贝属性，所以它不属于同态：

```
type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>
```

非同态类型本质上会创建新的属性，因此它们不会从它处拷贝属性修饰符。

由映射类型进行推断

现在你了解了如何包装一个类型的属性，那么接下来就是如何拆包。 其实这也非常容易：

```
function unproxify<T>(t: Proxify<T>): T {
    let result = {} as T;
    for (const k in t) {
        result[k] = t[k].get();
    }
    return result;
}

let originalProps = unproxify(proxyProps);
```

注意这个拆包推断只适用于同态的映射类型。 如果映射类型不是同态的，那么需要给拆包函数一个明确的类型参数。

介绍

自ECMAScript 2015起，`symbol`成为了一种新的原生类型，就像`number`和`string`一样。

`symbol`类型的值是通过`Symbol`构造函数创建的。

```
let sym1 = Symbol();

let sym2 = Symbol("key"); // 可选的字符串key
```

Symbols是不可改变且唯一的。

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

sym2 === sym3; // false, symbols是唯一的
```

像字符串一样，symbols也可以被用做对象属性的键。

```
let sym = Symbol();

let obj = {
    [sym]: "value"
};

console.log(obj[sym]); // "value"
```

Symbols也可以与计算出的属性名声明相结合来声明对象的属性和类成员。

```
const getClassNameSymbol = Symbol();

class C {
    [getClassNameSymbol](){
       return "C";
    }
}

let c = new C();
let className = c[getClassNameSymbol](); // "C"
```

众所周知的Symbols

除了用户定义的symbols，还有一些已经众所周知的内置symbols。 内置symbols用来表示语言内部的行为。

以下为这些symbols的列表：

`Symbol.hasInstance`

方法，会被 `instanceof` 运算符调用。构造器对象用来识别一个对象是否是其实例。

`Symbol.isConcatSpreadable`

布尔值，表示当在一个对象上调用 `Array.prototype.concat` 时，这个对象的数组元素是否可展开。

`Symbol.iterator`

方法，被 `for-of` 语句调用。返回对象的默认迭代器。

`Symbol.match`

方法，被 `String.prototype.match` 调用。正则表达式用来匹配字符串。

`Symbol.replace`

方法，被 `String.prototype.replace` 调用。正则表达式用来替换字符串中匹配的子串。

`Symbol.search`

方法，被 `String.prototype.search` 调用。正则表达式返回被匹配部分在字符串中的索引。

`Symbol.species`

函数值，为一个构造函数。用来创建派生对象。

`Symbol.split`

方法，被 `String.prototype.split` 调用。正则表达式来用分割字符串。

`Symbol.toPrimitive`

方法，被 `ToPrimitive` 抽象操作调用。把对象转换为相应的原始值。

`Symbol.toStringTag`

方法，被内置方法 `Object.prototype.toString` 调用。返回创建对象时默认的字符串描述。

`Symbol.unscopables`

对象，它自己拥有的属性会被 `with` 作用域排除在外。

可迭代性

当一个对象实现了 `Symbol.iterator` 属性时，我们认为它是可迭代的。 一些内置的类型
如 `Array` ， `Map` ， `Set` ， `String` ， `Int32Array` ， `Uint32Array` 等都已经实现了各自的 `Symbol.iterator` 。 对象上的 `Symbol.iterator` 函数负责返回供迭代的值。

`for..of` 语句

`for..of` 会遍历可迭代的对象，调用对象上的 `Symbol.iterator` 方法。 下面是在数组上使用 `for..of` 的简单例子：

```
let someArray = [1, "string", false];

for (let entry of someArray) {
    console.log(entry); // 1, "string", false
}
```

`for..of` vs. `for..in` 语句

`for..of` 和 `for..in` 均可迭代一个列表；但是用于迭代的值却不同， `for..in` 迭代的是对象的 键 的列表，而 `for..of` 则迭代对象的键对应的值。

下面的例子展示了两者之间的区别：

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

另一个区别是 `for..in` 可以操作任何对象；它提供了查看对象属性的一种方法。但是 `for..of` 关注于迭代对象的值。内置对象 `Map` 和 `Set` 已经实现了 `Symbol.iterator` 方法，让我们可以访问它们保存的值。

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
    console.log(pet); // "species"
}

for (let pet of pets) {
    console.log(pet); // "Cat", "Dog", "Hamster"
}
```

代码生成

目标为 ES5 和 ES3

当生成目标为ES5或ES3，迭代器只允许在 `Array` 类型上使用。 在非数组值上使用 `for..of` 语句会得到一个错误，就算这些非数组值已经实现了 `Symbol.iterator` 属性。

编译器会生成一个简单的 `for` 循环做为 `for..of` 循环，比如：

```
let numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}
```

生成的代码为：

```
var numbers = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
    var num = numbers[_i];
    console.log(num);
}
```

目标为 ECMAScript 2015 或更高

当目标为兼容ECMAScipt 2015的引擎时，编译器会生成相应引擎的 `for..of` 内置迭代器实现方式。

> 关于术语的一点说明: 请务必注意一点，TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”，这是为了与ECMAScript 2015里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

从ECMAScript 2015开始，JavaScript引入了模块的概念。TypeScript也沿用这个概念。

模块在其自身的作用域里执行，而不是在全局作用域里；这意味着定义在一个模块里的变量，函数，类等等在模块外部是不可见的，除非你明确地使用 `export` 形式之一导出它们。 相反，如果想使用其它模块导出的变量，函数，类，接口等的时候，你必须要导入它们，可以使用 `import` 形式之一。

模块是自声明的；两个模块之间的关系是通过在文件级别上使用imports和exports建立的。

模块使用模块加载器去导入其它的模块。 在运行时，模块加载器的作用是在执行此模块代码前去查找并执行这个模块的所有依赖。 大家最熟知的JavaScript模块加载器是服务于Node.js的CommonJS和服务于Web应用的Require.js。

TypeScript与ECMAScript 2015一样，任何包含顶级 `import` 或者 `export` 的文件都被当成一个模块。 相反地，如果一个文件不带有顶级的 `import` 或者 `export` 声明，那么它的内容被视为全局可见的（因此对模块也是可见的）。

导出

导出声明

任何声明（比如变量，函数，类，类型别名或接口）都能够通过添加 `export` 关键字来导出。

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

ZipCodeValidator.ts

```
export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

导出语句

导出语句很便利，因为我们可能需要对导出的部分重命名，所以上面的例子可以这样改写：

```
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

重新导出

我们经常会去扩展其它模块，并且只导出那个模块的部分内容。 重新导出功能并不会在当前模块导入那个模块或定义一个新的局部变量。

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// 导出原先的验证器但做了重命名
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";
```

或者一个模块可以包裹多个模块，并把他们导出的内容联合在一起通过语法： `export * from "module"` 。

AllValidators.ts

```
export * from "./StringValidator"; // exports interface StringValidator
export * from "./LettersOnlyValidator"; // exports class LettersOnlyValidator
export * from "./ZipCodeValidator";  // exports class ZipCodeValidator
```

导入

模块的导入操作与导出一样简单。 可以使用以下 `import` 形式之一来导入其它模块中的导出内容。

导入一个模块中的某个导出内容

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

可以对导入内容重命名

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

将整个模块导入到一个变量，并通过它来访问模块的导出部分

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

具有副作用的导入模块

尽管不推荐这么做，一些模块会设置一些全局状态供其它模块使用。 这些模块可能没有任何的导出或用户根本就不关注它的导出。 使用下面的方法来导入这类模块：

```
import "./my-module.js";
```

默认导出

每个模块都可以有一个 `default` 导出。 默认导出使用 `default` 关键字标记；并且一个模块只能够有一个 `default` 导出。 需要使用一种特殊的导入形式来导入 `default` 导出。

`default` 导出十分便利。 比如，像JQuery这样的类库可能有一个默认导出 `jQuery` 或 `$` ，并且我们基本上也会使用同样的名字 `jQuery` 或 `$` 导出JQuery。

JQuery.d.ts

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "JQuery";

$("button.continue").html( "Next Step..." );
```

类和函数声明可以直接被标记为默认导出。 标记为默认导出的类和函数的名字是可以省略的。

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.t
est(s);
    }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

或者

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

`default` 导出也可以是一个值

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

`export =` 和 `import = require()`

CommonJS和AMD都有一个 `exports` 对象的概念，它包含了一个模块的所有导出内容。

它们也支持把 `exports` 替换为一个自定义对象。 默认导出就好比这样一个功能；然而，它们却并不相互兼容。 TypeScript模块支持 `export =` 语法以支持传统的CommonJS和AMD的工作流模型。

`export =` 语法定义一个模块的导出对象。 它可以是类，接口，命名空间，函数或枚举。

若使用 `export =` 导出一个模块，则必须使用TypeScript的特定语法 `import module = require("module")` 来导入此模块。

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
  console.log(`"${ s }" - ${ validator.isAcceptable(s) ? "matche
s" : "does not match" }`);
});
```

生成模块代码

根据编译时指定的模块目标参数，编译器会生成相应的供Node.js (CommonJS)，Require.js (AMD)，UMD, SystemJS或ECMAScript 2015 native modules (ES6)模块加载系统使用的代码。 想要了解生成代码中 `define` ， `require` 和 `register` 的意义，请参考相应模块加载器的文档。

下面的例子说明了导入导出语句里使用的名字是怎么转换为相应的模块加载器代码的。

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, expor
ts, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
let mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports ===
"object") {
        let v = factory(require, exports); if (v !== undefined)
module.exports = v;
    }
    else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    let mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    let mod_1;
    let t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            }],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    }
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export let t = something + 1;
```

简单示例

下面我们来整理一下前面的验证器实现，每个模块只有一个命名的导出。

为了编译，我们必需要在命令行上指定一个模块目标。对于Node.js来说，使用 `--module commonjs` ；对于Require.js来说，使用 `--module amd` 。比如：

```
tsc --module commonjs Test.ts
```

编译完成后，每个模块会生成一个单独的 `.js` 文件。好比使用了reference标签，编译器会根据 `import` 语句编译相应的文件。

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";

const lettersRegexp = /^[A-Za-z]+$/;

export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable
(s) ? "matches" : "does not match" } ${ name }`);
    }
});
```

可选的模块加载和其它高级加载场景

有时候，你只想在某种条件下才加载某个模块。 在TypeScript里，使用下面的方式来实现它和其它的高级加载场景，我们可以直接调用模块加载器并且可以保证类型完全。

编译器会检测是否每个模块都会在生成的JavaScript中用到。 如果一个模块标识符只在类型注解部分使用，并且完全没有在表达式中使用时，就不会生成 `require` 这个模块的代码。 省略掉没有用到的引用对性能提升是很有益的，并同时提供了选择性加载模块的能力。

这种模式的核心是 `import id = require("...")` 语句可以让我们访问模块导出的类型。 模块加载器会被动态调用（通过 `require` ），就像下面 `if` 代码块里那样。 它利用了省略引用的优化，所以模块只在被需要时加载。 为了让这个模块工作，一定要注意 `import` 定义的标识符只能在表示类型处使用（不能在会转换成JavaScript的地方）。

为了确保类型安全性，我们可以使用 `typeof` 关键字。 `typeof` 关键字，当在表示类型的地方使用时，会得出一个类型值，这里就表示模块的类型。

示例：**Node.js**里的动态模块加载

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidat
or");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}
```

示例：**require.js**里的动态模块加载

```
declare function require(moduleNames: string[], onLoad: (...args
: any[]) => void): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zi
p) => {
        let validator = new ZipCodeValidator.ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}
```

示例：**System.js**里的动态模块加载

```
declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator:
typeof Zip) => {
        var x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}
```

使用其它的JavaScript库

要想描述非TypeScript编写的类库的类型，我们需要声明类库所暴露出的API。

我们叫它声明因为它不是“外部程序”的具体实现。 它们通常是在 `.d.ts` 文件里定义的。 如果你熟悉C/C++，你可以把它们当做 `.h` 文件。 让我们看一些例子。

外部模块

在Node.js里大部分工作是通过加载一个或多个模块实现的。 我们可以使用顶级的 `export` 声明来为每个模块都定义一个 `.d.ts` 文件，但最好还是写在一个大的 `.d.ts` 文件里。 我们使用与构造一个外部命名空间相似的方法，但是这里使用 `module` 关键字并且把名字用引号括起来，方便之后 `import` 。 例如：

node.d.ts (simplified excerpt)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, sla
shesDenoteHost?): Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export let sep: string;
}
```

现在我们可以 `/// <reference>` `node.d.ts` 并且使用 `import url = require("url");` 或 `import * as URL from "url"` 加载模块。

```
/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");
```

外部模块简写

假如你不想在使用一个新模块之前花时间去编写声明，你可以采用声明的简写形式以便能够快速使用它。

declarations.d.ts

```
declare module "hot-new-module";
```

简写模块里所有导出的类型将是 `any` 。

```
import x, {y} from "hot-new-module";
x(y);
```

模块声明通配符

某些模块加载器如SystemJS 和AMD支持导入非JavaScript内容。 它们通常会使用一个前缀或后缀来表示特殊的加载语法。 模块声明通配符可以用来表示这些情况。

```
declare module "*!text" {
    const content: string;
    export default content;
}
// Some do it the other way around.
declare module "json!*" {
    const value: any;
    export default value;
}
```

现在你可以就导入匹配 `"*!text"` 或 `"json!*"` 的内容了。

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD模块

有些模块被设计成兼容多个模块加载器，或者不使用模块加载器（全局变量）。 它们以UMD模块为代表。 这些库可以通过导入的形式或全局变量的形式访问。 例如：

math-lib.d.ts

```
export function isPrime(x: number): boolean;
export as namespace mathLib;
```

之后，这个库可以在某个模块里通过导入来使用：

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition fr
om inside a module
```

它同样可以通过全局变量的形式使用，但只能在某个脚本里。（脚本是指一个不带有导入或导出的文件。）

```
mathLib.isPrime(2);
```

创建模块结构指导

尽可能地在顶层导出

用户应该更容易地使用你模块导出的内容。嵌套层次过多会变得难以处理，因此仔细考虑一下如何组织你的代码。

从你的模块中导出一个命名空间就是一个增加嵌套的例子。虽然命名空间有时候有它们的用处，在使用模块的时候它们额外地增加了一层。这对用户来说是很不便的并且通常是多余的。

导出类的静态方法也有同样的问题 - 这个类本身就增加了一层嵌套。除非它能方便表述或便于清晰使用，否则请考虑直接导出一个辅助方法。

如果仅导出单个 `class` 或 `function`，使用 `export default`

就像“在顶层上导出”帮助减少用户使用的难度，一个默认的导出也能起到这个效果。如果一个模块就是为了导出特定的内容，那么你应该考虑使用一个默认导出。这会令模块的导入和使用变得些许简单。比如：

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

MyFunc.ts

```
export default function getThing() { return 'thing'; }
```

Consumer.ts

```
import t from "./MyClass";
import f from "./MyFunc";
let x = new t();
console.log(f());
```

对用户来说这是最理想的。他们可以随意命名导入模块的类型（本例为 `t` ）并且不需要多余的（.）来找到相关对象。

如果要导出多个对象，把它们放在顶层里导出

MyThings.ts

```
export class SomeType { /* ... */ }
export function someFunc() { /* ... */ }
```

相反地，当导入的时候：

明确地列出导入的名字

Consumer.ts

```
import { SomeType, SomeFunc } from "./MyThings";
let x = new SomeType();
let y = someFunc();
```

使用命名空间导入模式当你要导出大量内容的时候

MyLargeModule.ts

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

使用重新导出进行扩展

你可能经常需要去扩展一个模块的功能。 JS里常用的一个模式是JQuery那样去扩展原对象。 如我们之前提到的，模块不会像全局命名空间对象那样去合并。 推荐的方案是不要去改变原来的对象，而是导出一个新的实体来提供新的功能。

假设 `Calculator.ts` 模块里定义了一个简单的计算器实现。 这个模块同样提供了一个辅助函数来测试计算器的功能，通过传入一系列输入的字符串并在最后给出结果。

Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;
```

```
    protected processDigit(digit: string, currentValue: number)
{
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0"
.charCodeAt(0));
        }
    }

    protected processOperator(operator: string) {
        if (["+", "-", "*", "/"].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, r
ight: number): number {
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }

    private evaluate() {
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, t
his.memory, this.current);
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }

    public handleChar(char: string) {
        if (char === "=") {
            this.evaluate();
            return;
        }
```

```
        else {
            let value = this.processDigit(char, this.current);
            if (value !== undefined) {
                this.current = value;
                return;
            }
            else {
                let value = this.processOperator(char);
                if (value !== undefined) {
                    this.evaluate();
                    this.operator = value;
                    return;
                }
            }
        }
        throw new Error(`Unsupported input: '${char}'`);
    }

    public getResult() {
        return this.memory;
    }
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handleChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}
```

下面使用导出的 `test` 函数来测试计算器。

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";


let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

现在扩展它，添加支持输入其它进制（十进制以外），让我们来创建 `ProgrammerCalculator.ts` 。

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8"
, "9", "A", "B", "C", "D", "E", "F"];

    constructor(public base: number) {
        super();
        const maxBase = ProgrammerCalculator.digits.length;
        if (base <= 0 || base > maxBase) {
            throw new Error(`base has to be within 0 to ${maxBas
e} inclusive.`);
        }
    }

    protected processDigit(digit: string, currentValue: number)
{
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculat
or.digits.indexOf(digit);
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```

新的 `ProgrammerCalculator` 模块导出的API与原先的 `Calculator` 模块很相似，但却没有改变原模块里的对象。下面是测试ProgrammerCalculator类的代码：

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

模块里不要使用命名空间

当初次进入基于模块的开发模式时，可能总会控制不住要将导出包裹在一个命名空间里。 模块具有其自己的作用域，并且只有导出的声明才会在模块外部可见。 记住这点，命名空间在使用模块时几乎没什么价值。

在组织方面，命名空间对于在全局作用域内对逻辑上相关的对象和类型进行分组是很便利的。 例如，在C#里，你会从 `System.Collections` 里找到所有集合的类型。 通过将类型有层次地组织在命名空间里，可以方便用户找到与使用那些类型。 然而，模块本身已经存在于文件系统之中，这是必须的。 我们必须通过路径和文件名找到它们，这已经提供了一种逻辑上的组织形式。 我们可以创建 `/collections/generic/` 文件夹，把相应模块放在这里面。

命名空间对解决全局作用域里命名冲突来说是很重要的。 比如，你可以有一个 `My.Application.Customer.AddForm` 和 `My.Application.Order.AddForm` -- 两个类型的名字相同，但命名空间不同。 然而，这对于模块来说却不是一个问题。 在一个模块里，没有理由两个对象拥有同一个名字。 从模块的使用角度来说，使用者会挑出他们用来引用模块的名字，所以也没有理由发生重名的情况。

> 更多关于模块和命名空间的资料查看命名空间和模块

危险信号

以下均为模块结构上的危险信号。重新检查以确保你没有在对模块使用命名空间：

- 文件的顶层声明是 `export namespace Foo { ... }` （删除 `Foo` 并把所有内容向上层移动一层）
- 文件只有一个 `export class` 或 `export function` （考虑使用 `export default` ）
- 多个文件的顶层具有同样的 `export namespace Foo {` （不要以为这些会合并到一个 `Foo` 中！）

> 关于术语的一点说明: 请务必注意一点，TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”，这是为了与ECMAScript 2015里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

这篇文章描述了如何在TypeScript里使用命名空间（之前叫做“内部模块”）来组织你的代码。 就像我们在术语说明里提到的那样，“内部模块”现在叫做“命名空间”。 另外，任何使用 `module` 关键字来声明一个内部模块的地方都应该使用 `namespace` 关键字来替换。 这就避免了让新的使用者被相似的名称所迷惑。

第一步

我们先来写一段程序并将在整篇文章中都使用这个例子。 我们定义几个简单的字符串验证器，假设你会使用它们来验证表单里的用户输入或验证外部数据。

所有的验证器都放在一个文件里

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`'${ s }' ${ isMatch ? "matches" : "does not
 match" } '${ name }'.`);
    }
}
```

命名空间

随着更多验证器的加入，我们需要一种手段来组织代码，以便于在记录它们类型的同时还不用担心与其它对象产生命名冲突。 因此，我们把验证器包裹到一个命名空间内，而不是把它们放在全局命名空间下。

下面的例子里，把所有与验证器相关的类型都放到一个叫做 `Validation` 的命名空间里。 因为我们想让这些接口和类在命名空间之外也是可访问的，所以需要使用 `export` 。 相反的，变量 `lettersRegexp` 和 `numberRegexp` 是实现的细节，不需要导出，因此它们在命名空间外是不能访问的。 在文件末尾的测试代码里，由于是在命名空间之外访问，因此需要限定类型的名称，比如 `Validation.LettersOnlyValidator` 。

使用命名空间的验证器

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator
 {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];
```

```
// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {
};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator
();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable
(s) ? "matches" : "does not match" } ${ name }`);
    }
}
```

分离到多文件

当应用变得越来越大时，我们需要将代码分离到不同的文件中以便于维护。

多文件中的命名空间

现在，我们把 `Validation` 命名空间分割成多个文件。 尽管是不同的文件，它们仍是同一个命名空间，并且在使用的时候就如同它们在一个文件中定义的一样。 因为不同文件之间存在依赖关系，所以我们加入了引用标签来告诉编译器文件之间的关联。 我们的测试代码保持不变。

Validation.ts

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator
 {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {
};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator
();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable
(s) ? "matches" : "does not match" } ${ name }`);
    }
}
```

当涉及到多文件时，我们必须确保所有编译后的代码都被加载了。 我们有两种方式。

第一种方式，把所有的输入文件编译为一个输出文件，需要使用 `--outFile` 标记：

```
tsc --outFile sample.js Test.ts
```

编译器会根据源码里的引用标签自动地对输出进行排序。你也可以单独地指定每个文件。

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts Zi
pCodeValidator.ts Test.ts
```

第二种方式，我们可以编译每一个文件（默认方式），那么每个源文件都会对应生成一个JavaScript文件。 然后，在页面上通过 `<script>` 标签把所有生成的JavaScript文件按正确的顺序引进来，比如：

MyTestPage.html (excerpt)

```
    <script src="Validation.js" type="text/javascript" />
    <script src="LettersOnlyValidator.js" type="text/javascript"
 />
    <script src="ZipCodeValidator.js" type="text/javascript" />
    <script src="Test.js" type="text/javascript" />
```

别名

另一种简化命名空间操作的方法是使用 `import q = x.y.z` 给常用的对象起一个短的名字。 不要与用来加载模块的 `import x = require('name')` 语法弄混了，这里的语法是为指定的符号创建一个别名。 你可以用这种方法为任意标识符创建别名，也包括导入的模块中的对象。

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as "new Shapes.Polygons.
Square()"
```

注意，我们并没有使用 `require` 关键字，而是直接使用导入符号的限定名赋值。这与使用 `var` 相似，但它还适用于类型和导入的具有命名空间含义的符号。 重要的是，对于值来讲， `import` 会生成与原始符号不同的引用，所以改变别名的 `var` 值并不会影响原始变量的值。

使用其它的JavaScript库

为了描述不是用TypeScript编写的类库的类型，我们需要声明类库导出的API。 由于大部分程序库只提供少数的顶级对象，命名空间是用来表示它们的一个好办法。

我们称其为声明是因为它不是外部程序的具体实现。 我们通常在 `.d.ts` 里写这些声明。 如果你熟悉C/C++，你可以把它们当做 `.h` 文件。 让我们看一些例子。

外部命名空间

流行的程序库D3在全局对象 `d3` 里定义它的功能。 因为这个库通过一个 `<script>` 标签加载（不是通过模块加载器），它的声明文件使用内部模块来定义它的类型。 为了让TypeScript编译器识别它的类型，我们使用外部命名空间声明。 比如，我们可以像下面这样写：

D3.d.ts (部分摘录)

```
declare namespace D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```

> 关于术语的一点说明: 请务必注意一点，TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”，这是为了与ECMAScript 2015里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

介绍

这篇文章将概括介绍在TypeScript里使用模块与命名空间来组织代码的方法。 我们也会谈及命名空间和模块的高级使用场景，和在使用它们的过程中常见的陷阱。

查看模块章节了解关于模块的更多信息。 查看命名空间章节了解关于命名空间的更多信息。

使用命名空间

命名空间是位于全局命名空间下的一个普通的带有名字的JavaScript对象。 这令命名空间十分容易使用。 它们可以在多文件中同时使用，并通过 `--outFile` 结合在一起。 命名空间是帮你组织Web应用不错的方式，你可以把所有依赖都放在HTML页面的 `<script>` 标签里。

但就像其它的全局命名空间污染一样，它很难去识别组件之间的依赖关系，尤其是在大型的应用中。

使用模块

像命名空间一样，模块可以包含代码和声明。 不同的是模块可以声明它的依赖。

模块会把依赖添加到模块加载器上（例如CommonJs / Require.js）。 对于小型的JS应用来说可能没必要，但是对于大型应用，这一点点的花费会带来长久的模块化和可维护性上的便利。 模块也提供了更好的代码重用，更强的封闭性以及更好的使用工具进行优化。

对于Node.js应用来说，模块是默认并推荐的组织代码的方式。

从ECMAScript 2015开始，模块成为了语言内置的部分，应该会被所有正常的解释引擎所支持。 因此，对于新项目来说推荐使用模块做为组织代码的方式。

命名空间和模块的陷阱

这部分我们会描述常见的命名空间和模块的使用陷阱和如何去避免它们。

对模块使用 `/// <reference>`

一个常见的错误是使用 `/// <reference>` 引用模块文件，应该使用 `import` 。要理解这之间的区别，我们首先应该弄清编译器是如何根据 `import` 路径（例如， `import x from "...";` 或 `import x = require("...")` 里面的 `...` ，等等）来定位模块的类型信息的。

编译器首先尝试去查找相应路径下的 `.ts` ， `.tsx` 再或者 `.d.ts` 。 如果这些文件都找不到，编译器会查找外部模块声明。 回想一下，它们是在 `.d.ts` 文件里声明的。

- `myModules.d.ts`

```
// In a .d.ts file or .ts file that is not a module:
declare module "SomeModule" {
    export function fn(): string;
}
```

- `myOtherModule.ts`

```
/// <reference path="myModules.d.ts" />
import * as m from "SomeModule";
```

这里的引用标签指定了外来模块的位置。 这就是一些TypeScript例子中引用 `node.d.ts` 的方法。

不必要的命名空间

如果你想把命名空间转换为模块，它可能会像下面这个文件一件：

- `shapes.ts`

```
export namespace Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

顶层的模块 `Shapes` 包裹了 `Triangle` 和 `Square` 。 对于使用它的人来说这是令人迷惑和讨厌的：

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

TypeScript里模块的一个特点是不同的模块永远也不会在相同的作用域内使用相同的名字。 因为使用模块的人会为它们命名，所以完全没有必要把导出的符号包裹在一个命名空间里。

再次重申，不应该对模块使用命名空间，使用命名空间是为了提供逻辑分组和避免命名冲突。 模块文件本身已经是一个逻辑分组，并且它的名字是由导入这个模块的代码指定，所以没有必要为导出的对象增加额外的模块层。

下面是改进的例子：

- `shapes.ts`

```
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

模块的取舍

就像每个JS文件对应一个模块一样，TypeScript里模块文件与生成的JS文件也是一一对应的。 这会产生一种影响，根据你指定的目标模块系统的不同，你可能无法连接多个模块源文件。 例如当目标模块系统为 `commonjs` 或 `umd` 时，无法使用 `outFile` 选项，但是在TypeScript 1.8以上的版本能够使用 `outFile` 当目标为 `amd` 或 `system` 。

> 这节假设你已经了解了模块的一些基本知识 请阅读模块文档了解更多信息。

模块解析是指编译器在查找导入模块内容时所遵循的流程。 假设有一个导入语句 `import { a } from "moduleA"`; 为了去检查任何对 `a` 的使用，编译器需要准确的知道它表示什么，并且需要检查它的定义 `moduleA` 。

这时候，编译器会有个疑问“ `moduleA` 的结构是怎样的？”这听上去很简单，但 `moduleA` 可能在你写的某个 `.ts` / `.tsx` 文件里或者在你的代码所依赖的 `.d.ts` 里。

首先，编译器会尝试定位表示导入模块的文件。 编译器会遵循以下二种策略之一：Classic或Node。 这些策略会告诉编译器到哪里去查找 `moduleA` 。

如果上面的解析失败了并且模块名是非相对的（且是在 `"moduleA"` 的情况下），编译器会尝试定位一个外部模块声明。 我们接下来会讲到非相对导入。

最后，如果编译器还是不能解析这个模块，它会记录一个错误。 在这种情况下，错误可能为 `error TS2307: Cannot find module 'moduleA'.`

相对 vs. 非相对模块导入

根据模块引用是相对的还是非相对的，模块导入会以不同的方式解析。

相对导入是以 `/` ， `./` 或 `../` 开头的。 下面是一些例子：

- `import Entry from "./components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

所有其它形式的导入被当作非相对的。 下面是一些例子：

- `import * as $ from "jQuery";`
- `import { Component } from "@angular/core";`

相对导入在解析时是相对于导入它的文件，并且不能解析为一个外部模块声明。 你应该为你自己写的模块使用相对导入，这样能确保它们在运行时的相对位置。

非相对模块的导入可以相对于 `baseUrl` 或通过下文会讲到的路径映射来进行解析。 它们还可以被解析成外部模块声明。 使用非相对路径来导入你的外部依赖。

模块解析策略

共有两种可用的模块解析策略：Node和Classic。 你可以使用 `--moduleResolution` 标记来指定使用哪种模块解析策略。 若未指定，那么在使用了 `--module AMD | System | ES2015` 时的默认值为Classic，其它情况时则为Node。

Classic

这种策略在以前是TypeScript默认的解析策略。 现在，它存在的理由主要是为了向后兼容。

相对导入的模块是相对于导入它的文件进行解析的。 因此 `/root/src/folder/A.ts` 文件里的 `import { b } from "./moduleB"` 会使用下面的查找流程：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

对于非相对模块的导入，编译器则会从包含导入文件的目录开始依次向上级目录遍历，尝试定位匹配的声明文件。

比如：

有一个对 `moduleB` 的非相对导入 `import { b } from "moduleB"` ，它是在 `/root/src/folder/A.ts` 文件里，会以如下的方式来定位 `"moduleB"` ：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

Node

这个解析策略试图在运行时模仿Node.js模块解析机制。 完整的Node.js解析算法可以在Node.js module documentation找到。

Node.js如何解析模块

为了理解TypeScript编译依照的解析步骤，先弄明白Node.js模块是非常重要的。 通常，在Node.js里导入是通过 `require` 函数调用进行的。 Node.js会根据 `require` 的是相对路径还是非相对路径做出不同的行为。

相对路径很简单。 例如，假设有一个文件路径为 `/root/src/moduleA.js` ，包含了一个导入 `var x = require("./moduleB");` Node.js以下面的顺序解析这个导入：

1. 检查 `/root/src/moduleB.js` 文件是否存在。

2. 检查 `/root/src/moduleB` 目录是否包含一个 `package.json` 文件，且 `package.json` 文件指定了一个 `"main"` 模块。 在我们的例子里，如果Node.js发现文件 `/root/src/moduleB/package.json` 包含了 `{ "main": "lib/mainModule.js" }` ，那么Node.js会引用 `/root/src/moduleB/lib/mainModule.js` 。

3. 检查 `/root/src/moduleB` 目录是否包含一个 `index.js` 文件。 这个文件会被隐式地当作那个文件夹下的"main"模块。

你可以阅读Node.js文档了解更多详细信息：file modules 和 folder modules。

但是，非相对模块名的解析是个完全不同的过程。 Node会在一个特殊的文件夹 `node_modules` 里查找你的模块。 `node_modules` 可能与当前文件在同一级目录下，或者在上层目录里。 Node会向上级目录遍历，查找每个 `node_modules` 直到它找到要加载的模块。

还是用上面例子，但假设 `/root/src/moduleA.js` 里使用的是非相对路径导入 `var x = require("moduleB");` 。 Node则会以下面的顺序去解析 `moduleB` ，直到有一个匹配上。

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
3. `/root/src/node_modules/moduleB/index.js`

4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
6. `/root/node_modules/moduleB/index.js`

7. `/node_modules/moduleB.js`
8. `/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
9. `/node_modules/moduleB/index.js`

注意Node.js在步骤（4）和（7）会向上跳一级目录。

你可以阅读Node.js文档了解更多详细信息：loading modules from `node_modules` 。

TypeScript如何解析模块

TypeScript是模仿Node.js运行时的解析策略来在编译阶段定位模块定义文件。 因此，TypeScript在Node解析逻辑基础上增加了TypeScript源文件的扩展名（ `.ts` ， `.tsx` 和 `.d.ts` ）。 同时，TypeScript在 `package.json` 里使用字段 `"types"` 来表示类似 `"main"` 的意义 - 编译器会使用它来找到要使用的"main"定义文件。

比如，有一个导入语句 `import { b } from "./moduleB"` 在 `/root/src/moduleA.ts` 里，会以下面的流程来定位 `"./moduleB"` ：

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json` (如果指定了 `"types"` 属性)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

回想一下Node.js先查找 `moduleB.js` 文件，然后是合适的 `package.json` ，再之后是 `index.js` 。

类似地，非相对的导入会遵循Node.js的解析逻辑，首先查找文件，然后是合适的文件夹。 因此 `/root/src/moduleA.ts` 文件里的 `import { b } from "moduleB"` 会以下面的查找顺序解析：

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json` (如果指定了 `"types"` 属性)
5. `/root/src/node_modules/moduleB/index.ts`
6. `/root/src/node_modules/moduleB/index.tsx`
7. `/root/src/node_modules/moduleB/index.d.ts`

8. `/root/node_modules/moduleB.ts`
9. `/root/node_modules/moduleB.tsx`
10. `/root/node_modules/moduleB.d.ts`
11. `/root/node_modules/moduleB/package.json` (如果指定了 `"types"` 属性)
12. `/root/node_modules/moduleB/index.ts`
13. `/root/node_modules/moduleB/index.tsx`
14. `/root/node_modules/moduleB/index.d.ts`

15. `/node_modules/moduleB.ts`
16. `/node_modules/moduleB.tsx`
17. `/node_modules/moduleB.d.ts`
18. `/node_modules/moduleB/package.json` (如果指定了 `"types"` 属性)
19. `/node_modules/moduleB/index.ts`
20. `/node_modules/moduleB/index.tsx`
21. `/node_modules/moduleB/index.d.ts`

不要被这里步骤的数量吓到 - TypeScript只是在步骤（8）和（15）向上跳了两次目录。 这并不比Node.js里的流程复杂。

附加的模块解析标记

有时工程源码结构与输出结构不同。 通常是要经过一系统的构建步骤最后生成输出。 它们包括将 `.ts` 编译成 `.js` ，将不同位置的依赖拷贝至一个输出位置。 最终结果就是运行时的模块名与包含它们声明的源文件里的模块名不同。 或者最终输出文件里的模块路径与编译时的源文件路径不同了。

TypeScript编译器有一些额外的标记用来通知编译器在源码编译成最终输出的过程中都发生了哪个转换。

有一点要特别注意的是编译器不会进行这些转换操作；它只是利用这些信息来指导模块的导入。

Base URL

在利用AMD模块加载器的应用里使用 `baseUrl` 是常见做法，它要求在运行时模块都被放到了一个文件夹里。这些模块的源码可以在不同的目录下，但是构建脚本会将它们集中到一起。

设置 `baseUrl` 来告诉编译器到哪里去查找模块。所有非相对模块导入都会被当做相对于 `baseUrl` 。

*baseUrl*的值由以下两者之一决定：

- 命令行中*baseUrl*的值（如果给定的路径是相对的，那么将相对于当前路径进行计算）
- ‘tsconfig.json’里的*baseUrl*属性（如果给定的路径是相对的，那么将相对于‘tsconfig.json’路径进行计算）

注意相对模块的导入不会被设置的 `baseUrl` 所影响，因为它们总是相对于导入它们的文件。

阅读更多关于 `baseUrl` 的信息RequireJS和SystemJS。

路径映射

有时模块不是直接放在*baseUrl*下面。比如，充分 `"jquery"` 模块地导入，在运行时可能被解释为 `"node_modules/jquery/dist/jquery.slim.min.js"` 。加载器使用映射配置来将模块名映射到运行时的文件，查看RequireJs documentation和SystemJS documentation。

TypeScript编译器通过使用 `tsconfig.json` 文件里的 `"paths"` 来支持这样的声明映射。下面是一个如何指定 `jquery` 的 `"paths"` 的例子。

```
{
  "compilerOptions": {
    "baseUrl": ".", // This must be specified if "paths" is.
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"] // 此处映射是相对于"baseUrl"
    }
  }
}
```

请注意 `"paths"` 是相对于 `"baseUrl"` 进行解析。 如果 `"baseUrl"` 被设置成了除 `"."` 外的其它值，比如 `tsconfig.json` 所在的目录，那么映射必须要做相应的改变。 如果你在上例中设置了 `"baseUrl": "./src"` ，那么jquery应该映射到 `"../node_modules/jquery/dist/jquery"` 。

通过 `"paths"` 我们还可以指定复杂的映射，包括指定多个回退位置。 假设在一个工程配置里，有一些模块位于一处，而其它的则在另个的位置。 构建过程会将它们集中至一处。 工程结构可能如下：

```
projectRoot
├── folder1
│   ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')
│   └── file2.ts
├── generated
│   ├── folder1
│   └── folder2
│       └── file3.ts
└── tsconfig.json
```

相应的 `tsconfig.json` 文件如下：

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "*",
        "generated/*"
      ]
    }
  }
}
```

它告诉编译器所有匹配 `"*"` （所有的值）模式的模块导入会在以下两个位置查找：

1. `"*"` ：表示名字不发生改变，所以映射为 `<moduleName>` => `<baseUrl>/<moduleName>`
2. `"generated/*"` 表示模块名添加了“generated”前缀，所以映射为 `<moduleName>` => `<baseUrl>/generated/<moduleName>`

按照这个逻辑，编译器将会如下尝试解析这两个导入：

- 导入'folder1/file2'
    1. 匹配'*'模式且通配符捕获到整个名字。
    2. 尝试列表里的第一个替换：'*' -> `folder1/file2` 。
    3. 替换结果为非相对名 - 与*baseUrl*合并 -> `projectRoot/folder1/file2.ts` 。
    4. 文件存在。完成。
- 导入'folder2/file3'
    1. 匹配'*'模式且通配符捕获到整个名字。
    2. 尝试列表里的第一个替换：'*' -> `folder2/file3` 。
    3. 替换结果为非相对名 - 与*baseUrl*合并 -> `projectRoot/folder2/file3.ts` 。
    4. 文件不存在，跳到第二个替换。
    5. 第二个替换：'generated/*' -> `generated/folder2/file3` 。
    6. 替换结果为非相对名 - 与*baseUrl*合并 -> `projectRoot/generated/folder2/file3.ts` 。
    7. 文件存在。完成。

利用 `rootDirs` 指定虚拟目录

有时多个目录下的工程源文件在编译时会进行合并放在某个输出目录下。 这可以看做一些源目录创建了一个“虚拟”目录。

利用 `rootDirs` ，可以告诉编译器生成这个虚拟目录的*roots*； 因此编译器可以在“虚拟”目录下解析相对模块导入，就好像它们被合并在了一起一样。

比如，有下面的工程结构：

```
src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
        └── views
            └── template1.ts (imports './view2')
```

`src/views` 里的文件是用于控制UI的用户代码。 `generated/templates` 是UI模版，在构建时通过模版生成器自动生成。 构建中的一步会将 `/src/views` 和 `/generated/templates/views` 的输出拷贝到同一个目录下。 在运行时，视图可以假设它的模版与它同在一个目录下，因此可以使用相对导入 `"./template"` 。

可以使用 `"rootDirs"` 来告诉编译器。 `"rootDirs"` 指定了一个*roots*列表，列表里的内容会在运行时被合并。 因此，针对这个例子， `tsconfig.json` 如下：

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

每当编译器在某一 `rootDirs` 的子目录下发现了相对模块导入，它就会尝试从每一个 `rootDirs` 中导入。

`rootDirs` 的灵活性不仅仅局限于其指定了要在逻辑上合并的物理目录列表。它提供的数组可以包含任意数量的任何名字的目录，不论它们是否存在。这允许编译器以类型安全的方式处理复杂捆绑(bundles)和运行时的特性，比如条件引入和工程特定的加载器插件。

设想这样一个国际化的场景，构建工具自动插入特定的路径记号来生成针对不同区域的捆绑，比如将 `#{locale}` 做为相对模块路径 `./#{locale}/messages` 的一部分。在这个假定的设置下，工具会枚举支持的区域，将抽像的路径映射成 `./zh/messages` ， `./de/messages` 等。

假设每个模块都会导出一个字符串的数组。比如 `./zh/messages` 可能包含：

```
export default [
    "您好吗",
    "很高兴认识你"
];
```

利用 `rootDirs` 我们可以让编译器了解这个映射关系，从而也允许编译器能够安全地解析 `./#{locale}/messages` ，就算这个目录永远都不存在。比如，使用下面的 `tsconfig.json` ：

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/zh",
      "src/de",
      "src/#{locale}"
    ]
  }
}
```

编译器现在可以将 `import messages from './#{locale}/messages'` 解析为 `import messages from './zh/messages'` 用做工具支持的目的，并允许在开发时不必了解区域信息。

跟踪模块解析

如之前讨论，编译器在解析模块时可能访问当前文件夹外的文件。 这会导致很难诊断模块为什么没有被解析，或解析到了错误的位置。 通过 `--traceResolution` 启用编译器的模块解析跟踪，它会告诉我们在模块解析过程中发生了什么。

假设我们有一个使用了 `typescript` 模块的简单应用。 `app.ts` 里有一个这样的导入 `import * as ts from "typescript"` 。

```
│   tsconfig.json
├───node_modules
│   └───typescript
│       └───lib
│               typescript.d.ts
└───src
        app.ts
```

使用 `--traceResolution` 调用编译器。

```
tsc --traceResolution
```

输出结果如下：

```
======== Resolving module 'typescript' from 'src/app.ts'. ======
==
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'types' field './lib/typescript.d.ts' that re
ferences 'node_modules/typescript/lib/typescript.d.ts'.
File 'node_modules/typescript/lib/typescript.d.ts' exist - use i
t as a module resolution result.
======== Module name 'typescript' was successfully resolved to '
node_modules/typescript/lib/typescript.d.ts'. ========
```

需要留意的地方

- 导入的名字及位置

  > ======== Resolving module **'typescript'** from **'src/app.ts'**. ========

- 编译器使用的策略

  > Module resolution kind is not specified, using **'NodeJs'**.

- 从npm加载types

  > 'package.json' has **'types'** field './lib/typescript.d.ts' that references 'node_modules/typescript/lib/typescript.d.ts'.

- 最终结果

  > ======== Module name 'typescript' was **successfully resolved** to 'node_modules/typescript/lib/typescript.d.ts'. ========

使用 `--noResolve`

正常来讲编译器会在开始编译之前解析模块导入。 每当它成功地解析了对一个文件 `import` ，这个文件被会加到一个文件列表里，以供编译器稍后处理。

`--noResolve` 编译选项告诉编译器不要添加任何不是在命令行上传入的文件到编译列表。 编译器仍然会尝试解析模块，但是只要没有指定这个文件，那么它就不会被包含在内。

比如

app.ts

```
import * as A from "moduleA" // OK, moduleA passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.
```

```
tsc app.ts moduleA.ts --noResolve
```

使用 `--noResolve` 编译 `app.ts` ：

- 可能正确找到 `moduleA` ，因为它在命令行上指定了。
- 找不到 `moduleB` ，因为没有在命令行上传递。

常见问题

为什么在 `exclude` 列表里的模块还会被编译器使用

`tsconfig.json` 将文件夹转变一个“工程” 如果不指定任何 `“exclude”` 或 `“files”` ，文件夹里的所有文件包括 `tsconfig.json` 和所有的子目录都会在编译列表里。 如果你想利用 `“exclude”` 排除某些文件，甚至你想指定所有要编译的文件列表，请使用 `“files”` 。

有些是被 `tsconfig.json` 自动加入的。 它不会涉及到上面讨论的模块解析。 如果编译器识别出一个文件是模块导入目标，它就会加到编译列表里，不管它是否被排除了。

因此，要从编译列表中排除一个文件，你需要在排除它的同时，还要排除所有对它进行 `import` 或使用了 `/// <reference path="..." />` 指令的文件。

介绍

TypeScript中有些独特的概念可以在类型层面上描述JavaScript对象的模型。 这其中尤其独特的一个例子是“声明合并”的概念。 理解了这个概念，将有助于操作现有的JavaScript代码。 同时，也会有助于理解更多高级抽象的概念。

对本文件来讲，“声明合并”是指编译器将针对同一个名字的两个独立声明合并为单一声明。 合并后的声明同时拥有原先两个声明的特性。 任何数量的声明都可被合并；不局限于两个声明。

基础概念

TypeScript中的声明会创建以下三种实体之一：命名空间，类型或值。 创建命名空间的声明会新建一个命名空间，它包含了用（.）符号来访问时使用的名字。 创建类型的声明是：用声明的模型创建一个类型并绑定到给定的名字上。 最后，创建值的声明会创建在JavaScript输出中看到的值。

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

理解每个声明创建了什么，有助于理解当声明合并时有哪些东西被合并了。

合并接口

最简单也最常见的声明合并类型是接口合并。 从根本上说，合并的机制是把双方的成员放到一个同名的接口里。

```
interface Box {
    height: number;
    width: number;
}

interface Box {
    scale: number;
}

let box: Box = {height: 5, width: 6, scale: 10};
```

接口的非函数的成员应该是唯一的。 如果它们不是唯一的，那么它们必须是相同的类型。 如果两个接口中同时声明了同名的非函数成员且它们的类型不同，则编译器会报错。

对于函数成员，每个同名函数声明都会被当成这个函数的一个重载。 同时需要注意，当接口 `A` 与后来的接口 `A` 合并时，后面的接口具有更高的优先级。

如下例所示：

```
interface Cloner {
    clone(animal: Animal): Animal;
}

interface Cloner {
    clone(animal: Sheep): Sheep;
}

interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
}
```

这三个接口合并成一个声明：

```
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
    clone(animal: Sheep): Sheep;
    clone(animal: Animal): Animal;
}
```

注意每组接口里的声明顺序保持不变，但各组接口之间的顺序是后来的接口重载出现在靠前位置。

这个规则有一个例外是当出现特殊的函数签名时。 如果签名里有一个参数的类型是单一的字符串字面量（比如，不是字符串字面量的联合类型），那么它将会被提升到重载列表的最顶端。

比如，下面的接口会合并到一起：

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

合并后的 `Document` 将会像下面这样：

```
interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

合并命名空间

与接口相似，同名的命名空间也会合并其成员。 命名空间会创建出命名空间和值，我们需要知道这两者都是怎么合并的。

对于命名空间的合并，模块导出的同名接口进行合并，构成单一命名空间内含合并后的接口。

对于命名空间里值的合并，如果当前已经存在给定名字的命名空间，那么后来的命名空间的导出成员会被加到已经存在的那个模块里。

`Animals` 声明合并示例：

```
namespace Animals {
    export class Zebra { }
}

namespace Animals {
    export interface Legged { numberOfLegs: number; }
    export class Dog { }
}
```

等同于：

```
namespace Animals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}
```

除了这些合并外，你还需要了解非导出成员是如何处理的。 非导出成员仅在其原有的（合并前的）命名空间内可见。这就是说合并之后，从其它命名空间合并进来的成员无法访问非导出成员。

下例提供了更清晰的说明：

```
namespace Animal {
    let haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

namespace Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // <-- error, haveMuscles is not vi
sible here
    }
}
```

因为 `haveMuscles` 并没有导出，只有 `animalsHaveMuscles` 函数共享了原始未合并的命名空间可以访问这个变量。 `doAnimalsHaveMuscles` 函数虽是合并命名空间的一部分，但是访问不了未导出的成员。

命名空间与类和函数和枚举类型合并

命名空间可以与其它类型的声明进行合并。 只要命名空间的定义符合将要合并类型的定义。合并结果包含两者的声明类型。 TypeScript使用这个功能去实现一些JavaScript里的设计模式。

合并命名空间和类

这让我们可以表示内部类。

```
class Album {
    label: Album.AlbumLabel;
}
namespace Album {
    export class AlbumLabel { }
}
```

合并规则与上面 `合并命名空间` 小节里讲的规则一致，我们必须导出 `AlbumLabel` 类，好让合并的类能访问。 合并结果是一个类并带有一个内部类。 你也可以使用命名空间为类增加一些静态属性。

除了内部类的模式，你在JavaScript里，创建一个函数稍后扩展它增加一些属性也是很常见的。 TypeScript使用声明合并来达到这个目的并保证类型安全。

```
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}

namespace buildLabel {
    export let suffix = "";
    export let prefix = "Hello, ";
}

alert(buildLabel("Sam Smith"));
```

相似的，命名空间可以用来扩展枚举型：

```
enum Color {
    red = 1,
    green = 2,
    blue = 4
}

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

非法的合并

TypeScript并非允许所有的合并。目前，类不能与其它类或变量合并。想要了解如何模仿类的合并，请参考TypeScript的混入。

模块扩展

虽然JavaScript不支持合并，但你可以为导入的对象打补丁以更新它们。让我们考察一下这个玩具性的示例：

```
// observable.js
export class Observable<T> {
    // ... implementation left as an exercise for the reader ...
}

// map.js
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}
```

它也可以很好地工作在TypeScript中， 但编译器对 `Observable.prototype.map` 一无所知。 你可以使用扩展模块来将它告诉编译器：

```
// observable.ts stays the same
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}


// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map(x => x.toFixed());
```

模块名的解析和用 `import` / `export` 解析模块标识符的方式是一致的。 更多信息请参考 Modules。 当这些声明在扩展中合并时，就好像在原始位置被声明了一样。 但是，你不能在扩展中声明新的顶级声明－仅可以扩展模块中已经存在的声明。

全局扩展

你也以在模块内部添加声明到全局作用域中。

```
// observable.ts
export class Observable<T> {
    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}
```

全局扩展与模块扩展的行为和限制是相同的。

> 本页面被移动到书写声明文件页

介绍

JSX是一种嵌入式的类似XML的语法。 它可以被转换成合法的JavaScript，尽管转换的语义是依据不同的实现而定的。 JSX因React框架而流行，但也存在其它的实现。 TypeScript支持内嵌，类型检查以及将JSX直接编译为JavaScript。

基本用法

想要使用JSX必须做两件事：

1. 给文件一个 `.tsx` 扩展名
2. 启用 `jsx` 选项

TypeScript具有三种JSX模式： `preserve` ， `react` 和 `react-native` 。 这些模式只在代码生成阶段起作用 - 类型检查并不受影响。 在 `preserve` 模式下生成代码中会保留JSX以供后续的转换操作使用（比如：Babel）。 另外，输出文件会带有 `.jsx` 扩展名。 `react` 模式会生成 `React.createElement` ，在使用前不需要再进行转换操作了，输出文件的扩展名为 `.js` 。 `react-native` 相当于 `preserve` ，它也保留了所有的JSX，但是输出文件的扩展名是 `.js` 。

模式	输入	输出	输出文件扩展名
`preserve`	`<div />`	`<div />`	`.jsx`
`react`	`<div />`	`React.createElement("div")`	`.js`
`react-native`	`<div />`	`<div />`	`.js`

你可以通过在命令行里使用 `--jsx` 标记或tsconfig.json里的选项来指定模式。

> 注意： *`React`* 标识符是写死的硬编码，所以你必须保证*React*（大写的*R*）是可用的。

`as` 操作符

回想一下怎么写类型断言：

```
var foo = <foo>bar;
```

这里断言 `bar` 变量是 `foo` 类型的。 因为TypeScript也使用尖括号来表示类型断言，在结合JSX的语法后将带来解析上的困难。因此，TypeScript在 `.tsx` 文件里禁用了使用尖括号的类型断言。

由于不能够在 `.tsx` 文件里使用上述语法，因此我们应该使用另一个类型断言操作符： `as` 。 上面的例子可以很容易地使用 `as` 操作符改写：

```
var foo = bar as foo;
```

`as` 操作符在 `.ts` 和 `.tsx` 里都可用，并且与尖括号类型断言行为是等价的。

类型检查

为了理解JSX的类型检查，你必须首先理解固有元素与基于值的元素之间的区别。 假设有这样一个JSX表达式 `<expr />` ， `expr` 可能引用环境自带的某些东西（比如，在DOM环境里的 `div` 或 `span` ）或者是你自定义的组件。 这是非常重要的，原因有如下两点：

1. 对于React，固有元素会生成字符串（ `React.createElement("div")` ）， 然而由你自定义的组件却不会生成（ `React.createElement(MyComponent)` ）。
2. 传入JSX元素里的属性类型的查找方式不同。 固有元素属性本身就支持，然而自定义的组件会自己去指定它们具有哪个属性。

TypeScript使用与React相同的规范 来区别它们。 固有元素总是以一个小写字母开头，基于值的元素总是以一个大写字母开头。

固有元素

固有元素使用特殊的接口 `JSX.IntrinsicElements` 来查找。 默认地，如果这个接口没有指定，会全部通过，不对固有元素进行类型检查。 然而，如果这个接口存在，那么固有元素的名字需要在 `JSX.IntrinsicElements` 接口的属性里查找。

例如：

```
declare namespace JSX {
    interface IntrinsicElements {
        foo: any
    }
}

<foo />; // 正确
<bar />; // 错误
```

在上例中，`<foo />` 没有问题，但是 `<bar />` 会报错，因为它没在 `JSX.IntrinsicElements` 里指定。

> 注意：你也可以在 `JSX.IntrinsicElements` 上指定一个用来捕获所有字符串索引：
>
> ```
> declare namespace JSX {
>    interface IntrinsicElements {
>        [elemName: string]: any;
>    }
> }
> ```

基于值的元素

基于值的元素会简单的在它所在的作用域里按标识符查找。

```
import MyComponent from "./myComponent";

<MyComponent />; // 正确
<SomeOtherComponent />; // 错误
```

有两种方式可以定义基于值的元素：

1. 无状态函数组件 (SFC)
2. 类组件

由于这两种基于值的元素在JSX表达式里无法区分，因此TypeScript首先会尝试将表达式做为无状态函数组件进行解析。如果解析成功，那么TypeScript就完成了表达式到其声明的解析操作。如果按照无状态函数组件解析失败，那么TypeScript会继续尝试以类组件的形式进行解析。如果依旧失败，那么将输出一个错误。

无状态函数组件

正如其名，组件被定义成JavaScript函数，它的第一个参数是 `props` 对象。TypeScript会强制它的返回值可以赋值给 `JSX.Element` 。

```
interface FooProp {
  name: string;
  X: number;
  Y: number;
}

declare function AnotherComponent(prop: {name: string});
function ComponentFoo(prop: FooProp) {
  return <AnotherComponent name=prop.name />;
}

const Button = (prop: {value: string}, context: { color: string
}) => <button>
```

由于无状态函数组件是简单的JavaScript函数，所以我们还可以利用函数重载。

```
interface ClickableProps {
  children: JSX.Element[] | JSX.Element
}

interface HomeProps extends ClickableProps {
  home: JSX.Element;
}

interface SideProps extends ClickableProps {
  side: JSX.Element | string;
}

function MainButton(prop: HomeProps): JSX.Element;
function MainButton(prop: SideProps): JSX.Element {
  ...
}
```

类组件

我们可以定义类组件的类型。 然而，我们首先最好弄懂两个新的术语：元素类的类型和元素实例的类型。

现在有 `<Expr />` ，元素类的类型为 `Expr` 的类型。 所以在上面的例子里，如果 `MyComponent` 是ES6的类，那么类类型就是类的构造函数和静态部分。 如果 `MyComponent` 是个工厂函数，类类型为这个函数。

一旦建立起了类类型，实例类型由类构造器或调用签名（如果存在的话）的返回值的联合构成。 再次说明，在ES6类的情况下，实例类型为这个类的实例的类型，并且如果是工厂函数，实例类型为这个函数返回值类型。

```
class MyComponent {
  render() {}
}

// 使用构造签名
var myComponent = new MyComponent();

// 元素类的类型 => MyComponent
// 元素实例的类型 => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {
    }
  }
}

// 使用调用签名
var myComponent = MyFactoryFunction();

// 元素类的类型 => FactoryFunction
// 元素实例的类型 => { render: () => void }
```

元素的实例类型很有趣，因为它必须赋值给 `JSX.ElementClass` 或抛出一个错误。默认的 `JSX.ElementClass` 为 `{}` ，但是它可以被扩展用来限制JSX的类型以符合相应的接口。

```
declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // 正确
<MyFactoryFunction />; // 正确

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // 错误
<NotAValidFactoryFunction />; // 错误
```

属性类型检查

属性类型检查的第一步是确定元素属性类型。 这在固有元素和基于值的元素之间稍有不同。

对于固有元素，这是 `JSX.IntrinsicElements` 属性的类型。

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { bar?: boolean }
  }
}

// `foo`的元素属性类型为`{bar?: boolean}`
<foo bar />;
```

对于基于值的元素，就稍微复杂些。 它取决于先前确定的在元素实例类型上的某个属性的类型。 至于该使用哪个属性来确定类型取决于 `JSX.ElementAttributesProperty` 。 它应该使用单一的属性来定义。 这个属性名之后会被使用。 TypeScript 2.8，如果未指定 `JSX.ElementAttributesProperty` ，那么将使用类元素构造函数或SFC调用的第一个参数的类型。

```
declare namespace JSX {
  interface ElementAttributesProperty {
    props; // 指定用来使用的属性名
  }
}

class MyComponent {
  // 在元素实例类型上指定属性
  props: {
    foo?: string;
  }
}

// `MyComponent`的元素属性类型为`{foo?: string}`
<MyComponent foo="bar" />
```

元素属性类型用于的JSX里进行属性的类型检查。 支持可选属性和必须属性。

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { requiredProp: string; optionalProp?: number }
  }
}

<foo requiredProp="bar" />; // 正确
<foo requiredProp="bar" optionalProp={0} />; // 正确
<foo />; // 错误, 缺少 requiredProp
<foo requiredProp={0} />; // 错误, requiredProp 应该是字符串
<foo requiredProp="bar" unknownProp />; // 错误, unknownProp 不存在

<foo requiredProp="bar" some-unknown-prop />; // 正确, `some-unknown-prop`不是个合法的标识符
```

> 注意：如果一个属性名不是个合法的JS标识符（像 `data-*` 属性），并且它没出现在元素属性类型里时不会当做一个错误。

另外，JSX还会使用 `JSX.IntrinsicAttributes` 接口来指定额外的属性，这些额外的属性通常不会被组件的props或arguments使用 - 比如React里的 `key` 。还有， `JSX.IntrinsicClassAttributes<T>` 泛型类型也可以用来做同样的事情。这里的泛型参数表示类实例类型。在React里，它用来允许 `Ref<T>` 类型上的 `ref` 属性。通常来讲，这些接口上的所有属性都是可选的，除非你想要用户在每个JSX标签上都提供一些属性。

延展操作符也可以使用：

```
var props = { requiredProp: 'bar' };
<foo {...props} />; // 正确

var badProps = {};
<foo {...badProps} />; // 错误
```

子孙类型检查

从TypeScript 2.3开始，我们引入了*children*类型检查。*children*是元素属性*(attribute)*类型的一个特殊属性(property)，子*JSXExpression*将会被插入到属性里。与使用 `JSX.ElementAttributesProperty` 来决定*props*名类似，我们可以利用 `JSX.ElementChildrenAttribute` 来决定*children*名。`JSX.ElementChildrenAttribute` 应该被声明在单一的属性(property)里。

```
declare namespace JSX {
  interface ElementChildrenAttribute {
    children: {};  // specify children name to use
  }
}
```

如不特殊指定子孙的类型，我们将使用React typings里的默认类型。

```
<div>
  <h1>Hello</h1>
</div>;

<div>
  <h1>Hello</h1>
  World
</div>;

const CustomComp = (props) => <div>props.children</div>
<CustomComp>
  <div>Hello World</div>
  {"This is just a JS expression..." + 1000}
</CustomComp>
```

```
interface PropsType {
  children: JSX.Element
  name: string
}

class Component extends React.Component<PropsType, {}> {
  render() {
    return (
      <h2>
        {this.props.children}
      </h2>
    )
  }
}

// OK
<Component>
  <h1>Hello World</h1>
</Component>

// Error: children is of type JSX.Element not array of JSX.Eleme
nt
<Component>
  <h1>Hello World</h1>
  <h2>Hello World</h2>
</Component>

// Error: children is of type JSX.Element not array of JSX.Eleme
nt or string.
<Component>
  <h1>Hello</h1>
  World
</Component>
```

JSX结果类型

默认地JSX表达式结果的类型为 `any` 。 你可以自定义这个类型，通过指定 `JSX.Element` 接口。 然而，不能够从接口里检索元素，属性或JSX的子元素的类型信息。 它是一个黑盒。

嵌入的表达式

JSX允许你使用 `{ }` 标签来内嵌表达式。

```
var a = <div>
  {['foo', 'bar'].map(i => <span>{i / 2}</span>)}
</div>
```

上面的代码产生一个错误，因为你不能用数字来除以一个字符串。 输出如下，若你使用了 `preserve` 选项：

```
var a = <div>
  {['foo', 'bar'].map(function (i) { return <span>{i / 2}</span>
; })}
</div>
```

React整合

要想一起使用JSX和React，你应该使用React类型定义。 这些类型声明定义了 `JSX` 合适命名空间来使用React。

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent foo="bar" />; // 正确
<MyComponent foo={0} />; // 错误
```

工厂函数

`jsx: react` 编译选项使用的工厂函数是可以配置的。可以使用 `jsxFactory` 命令行选项，或内联的 `@jsx` 注释指令在每个文件上设置。比如，给 `createElement` 设置 `jsxFactory` ，`</div>` 会使用 `createElement("div")` 来生成，而不是 `React.createElement("div")` 。

注释指令可以像下面这样使用（在TypeScript 2.8里）：

```
import preact = require("preact");
/* @jsx preact.h */
const x = </div>;
```

生成：

```
const preact = require("preact");
const x = preact.h("div", null);
```

工厂函数的选择同样会影响 `JSX` 命名空间的查找（类型检查）。如果工厂函数使用 `React.createElement` 定义（默认），编译器会先检查 `React.JSX` ，之后才检查全局的 `JSX` 。如果工厂函数定义为 `h` ，那么在检查全局的 `JSX` 之前先检

查 `h.JSX` 。

介绍

随着TypeScript和ES6里引入了类，在一些场景下我们需要额外的特性来支持标注或修改类及其成员。 装饰器（Decorators）为我们在类的声明及成员上通过元编程语法添加标注提供了一种方式。 Javascript里的装饰器目前处在建议征集的第二阶段，但在TypeScript里已做为一项实验性特性予以支持。

> 注意 装饰器是一项实验性特性，在未来的版本中可能会发生改变。

若要启用实验性的装饰器特性，你必须在命令行或`tsconfig.json`里启用`experimentalDecorators`编译器选项：

命令行:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

装饰器

装饰器是一种特殊类型的声明，它能够被附加到类声明，方法，访问符，属性或参数上。 装饰器使用`@expression`这种形式，`expression`求值后必须为一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入。

例如，有一个`@sealed`装饰器，我们会这样定义`sealed`函数：

```
function sealed(target) {
    // do something with "target" ...
}
```

> 注意　后面类装饰器小节里有一个更加详细的例子。

装饰器工厂

如果我们要定制一个修饰器如何应用到一个声明上，我们得写一个装饰器工厂函数。 装饰器工厂就是一个简单的函数，它返回一个表达式，以供装饰器在运行时调用。

我们可以通过下面的方式来写一个装饰器工厂函数：

```
function color(value: string) { // 这是一个装饰器工厂
    return function (target) { //  这是装饰器
        // do something with "target" and "value"...
    }
}
```

> 注意　下面方法装饰器小节有一个更加详细的例子。

装饰器组合

多个装饰器可以同时应用到一个声明上，就像下面的示例：

- 书写在同一行上：

```
@f @g x
```

- 书写在多行上：

```
@f
@g
x
```

当多个装饰器应用于一个声明上，它们求值方式与复合函数相似。在这个模型下，当复合f和g时，复合的结果$(f \circ g)(x)$等同于$f(g(x))$。

同样的，在TypeScript里，当多个装饰器应用在一个声明上时会进行如下步骤的操作：

1. 由上至下依次对装饰器表达式求值。
2. 求值的结果会被当作函数，由下至上依次调用。

如果我们使用装饰器工厂的话，可以通过下面的例子来观察它们求值的顺序：

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey: string, descriptor: Pr
opertyDescriptor) {
        console.log("f(): called");
    }
}

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor: Pr
opertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

在控制台里会打印出如下结果：

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

装饰器求值

类中不同声明上的装饰器将按以下规定的顺序应用：

1. 参数装饰器，然后依次是方法装饰器，访问符装饰器，或属性装饰器应用到每个实例成员。
2. 参数装饰器，然后依次是方法装饰器，访问符装饰器，或属性装饰器应用到每个静态成员。
3. 参数装饰器应用到构造函数。
4. 类装饰器应用到类。

类装饰器

类装饰器在类声明之前被声明（紧靠着类声明）。 类装饰器应用于类构造函数，可以用来监视，修改或替换类定义。 类装饰器不能用在声明文件中(`.d.ts`)，也不能用在任何外部上下文中（比如 `declare` 的类）。

类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。

如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明。

> 注意 如果你要返回一个新的构造函数，你必须注意处理好原来的原型链。 在运行时的装饰器调用逻辑中不会为你做这些。

下面是使用类装饰器(`@sealed`)的例子，应用在 `Greeter` 类：

```
@sealed
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

我们可以这样定义 `@sealed` 装饰器：

```
function sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}
```

当 `@sealed` 被执行的时候，它将密封此类的构造函数和原型。(注：参见 Object.seal)

下面是一个重载构造函数的例子。

```
function classDecorator<T extends {new(...args:any[]):{}}>(constructor:T) {
    return class extends constructor {
        newProperty = "new property";
        hello = "override";
    }
}

@classDecorator
class Greeter {
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }
}

console.log(new Greeter("world"));
```

方法装饰器

方法装饰器声明在一个方法的声明之前（紧靠着方法声明）。 它会被应用到方法的属性描述符上，可以用来监视，修改或者替换方法定义。 方法装饰器不能用在声明文件(`.d.ts`)，重载或者任何外部上下文（比如 `declare` 的类）中。

方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。

2. 成员的名字。
3. 成员的属性描述符。

> 注意　如果代码输出目标版本小于 `ES5`，属性描述符将会是 `undefined`。

如果方法装饰器返回一个值，它会被用作方法的属性描述符。

> 注意　如果代码输出目标版本小于 `ES5` 返回值会被忽略。

下面是一个方法装饰器（`@enumerable`）的例子，应用于 `Greeter` 类的方法上：

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }

    @enumerable(false)
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

我们可以用下面的函数声明来定义 `@enumerable` 装饰器：

```
function enumerable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        descriptor.enumerable = value;
    };
}
```

这里的 `@enumerable(false)` 是一个装饰器工厂。 当装饰器 `@enumerable(false)` 被调用时，它会修改属性描述符的 `enumerable` 属性。

访问器装饰器

访问器装饰器声明在一个访问器的声明之前（紧靠着访问器声明）。 访问器装饰器应用于访问器的属性描述符并且可以用来监视，修改或替换一个访问器的定义。 访问器装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如 `declare` 的类）里。

> 注意 TypeScript不允许同时装饰一个成员的 `get` 和 `set` 访问器。取而代之的是，一个成员的所有装饰的必须应用在文档顺序的第一个访问器上。这是因为，在装饰器应用于一个属性描述符时，它联合了 `get` 和 `set` 访问器，而不是分开声明的。

访问器装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的属性描述符。

> 注意 如果代码输出目标版本小于 `ES5` ，*Property Descriptor*将会是 `undefined` 。

如果访问器装饰器返回一个值，它会被用作方法的属性描述符。

> 注意 如果代码输出目标版本小于 `ES5` 返回值会被忽略。

下面是使用了访问器装饰器（ `@configurable` ）的例子，应用于 `Point` 类的成员上：

```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }

    @configurable(false)
    get x() { return this._x; }

    @configurable(false)
    get y() { return this._y; }
}
```

我们可以通过如下函数声明来定义 `@configurable` 装饰器：

```
function configurable(value: boolean) {
    return function (target: any, propertyKey: string, descripto
r: PropertyDescriptor) {
        descriptor.configurable = value;
    };
}
```

属性装饰器

属性装饰器声明在一个属性声明之前（紧靠着属性声明）。 属性装饰器不能用在声明文件中（.d.ts），或者任何外部上下文（比如 `declare` 的类）里。

属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。

> 注意 属性描述符不会做为参数传入属性装饰器，这与TypeScript是如何初始化属性装饰器的有关。 因为目前没有办法在定义一个原型对象的成员时描述一个实例属性，并且没办法监视或修改一个属性的初始化方法。返回值也会被忽略。 因此，属性描述符只能用来监视类中是否声明了某个名字的属性。

如果访问符装饰器返回一个值，它会被用作方法的属性描述符。

我们可以用它来记录这个属性的元数据，如下例所示：

```
class Greeter {
    @format("Hello, %s")
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        let formatString = getFormat(this, "greeting");
        return formatString.replace("%s", this.greeting);
    }
}
```

然后定义 `@format` 装饰器和 `getFormat` 函数：

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
    return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
    return Reflect.getMetadata(formatMetadataKey, target, proper
tyKey);
}
```

这个 `@format("Hello, %s")` 装饰器是个 装饰器工厂。 当 `@format("Hello, %s")` 被调用时，它添加一条这个属性的元数据，通过 `reflect-metadata` 库里的 `Reflect.metadata` 函数。 当 `getFormat` 被调用时，它读取格式的元数据。

> 注意　这个例子需要使用 `reflect-metadata` 库。 查看元数据了解 `reflect-metadata` 库更详细的信息。

参数装饰器

参数装饰器声明在一个参数声明之前（紧靠着参数声明）。 参数装饰器应用于类构造函数或方法声明。 参数装饰器不能用在声明文件（.d.ts），重载或其它外部上下文（比如 `declare` 的类）里。

参数装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 参数在函数参数列表中的索引。

> 注意　参数装饰器只能用来监视一个方法的参数是否被传入。

参数装饰器的返回值会被忽略。

下例定义了参数装饰器（ `@required` ）并应用于 `Greeter` 类方法的一个参数：

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    @validate
    greet(@required name: string) {
        return "Hello " + name + ", " + this.greeting;
    }
}
```

然后我们使用下面的函数定义 `@required` 和 `@validate` 装饰器：

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol,
parameterIndex: number) {
    let existingRequiredParameters: number[] = Reflect.getOwnMet
adata(requiredMetadataKey, target, propertyKey) || [];
    existingRequiredParameters.push(parameterIndex);
    Reflect.defineMetadata(requiredMetadataKey, existingRequired
Parameters, target, propertyKey);
}

function validate(target: any, propertyName: string, descriptor:
 TypedPropertyDescriptor<Function>) {
    let method = descriptor.value;
    descriptor.value = function () {
        let requiredParameters: number[] = Reflect.getOwnMetadat
a(requiredMetadataKey, target, propertyName);
        if (requiredParameters) {
            for (let parameterIndex of requiredParameters) {
                if (parameterIndex >= arguments.length || argume
nts[parameterIndex] === undefined) {
                    throw new Error("Missing required argument."
);
                }
            }
        }

        return method.apply(this, arguments);
    }
}
```

`@required` 装饰器添加了元数据实体把参数标记为必需的。 `@validate` 装饰器把 `greet` 方法包裹在一个函数里在调用原先的函数前验证函数参数。

> 注意　这个例子使用了 `reflect-metadata` 库。查看元数据了解 `reflect-metadata` 库的更多信息。

元数据

一些例子使用了 `reflect-metadata` 库来支持实验性的metadata API。 这个库还不是ECMAScript (JavaScript)标准的一部分。 然而，当装饰器被ECMAScript官方标准采纳后，这些扩展也将被推荐给ECMAScript以采纳。

你可以通过npm安装这个库：

```
npm i reflect-metadata --save
```

TypeScript支持为带有装饰器的声明生成元数据。 你需要在命令行或 `tsconfig.json` 里启用 `emitDecoratorMetadata` 编译器选项。

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadat
a
```

tsconfig.json:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    }
}
```

当启用后，只要 `reflect-metadata` 库被引入了，设计阶段添加的类型信息可以在运行时使用。

如下例所示：

```
import "reflect-metadata";

class Point {
    x: number;
    y: number;
}

class Line {
    private _p0: Point;
    private _p1: Point;

    @validate
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}

function validate<T>(target: any, propertyKey: string, descripto
r: TypedPropertyDescriptor<T>) {
    let set = descriptor.set;
    descriptor.set = function (value: T) {
        let type = Reflect.getMetadata("design:type", target, pr
opertyKey);
        if (!(value instanceof type)) {
            throw new TypeError("Invalid type.");
        }
        set(value);
    }
}
```

TypeScript编译器可以通过 `@Reflect.metadata` 装饰器注入设计阶段的类型信息。你可以认为它相当于下面的TypeScript：

```
class Line {
    private _p0: Point;
    private _p1: Point;

    @validate
    @Reflect.metadata("design:type", Point)
    set p0(value: Point) { this._p0 = value; }
    get p0() { return this._p0; }

    @validate
    @Reflect.metadata("design:type", Point)
    set p1(value: Point) { this._p1 = value; }
    get p1() { return this._p1; }
}
```

> 注意　装饰器元数据是个实验性的特性并且可能在以后的版本中发生破坏性的改变（breaking changes）。

介绍

除了传统的面向对象继承方式，还流行一种通过可重用组件创建类的方式，就是联合另一个简单类的代码。 你可能在Scala等语言里对mixins及其特性已经很熟悉了，但它在JavaScript中也是很流行的。

混入示例

下面的代码演示了如何在TypeScript里使用混入。 后面我们还会解释这段代码是怎么工作的。

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}

class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " + th
is.isDisposed), 500);
    }
```

```
    interact() {
        this.activate();
    }

    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activatable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable]);

let smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);

////////////////////////////////////////
// In your runtime library somewhere
////////////////////////////////////////

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(n
ame => {
            derivedCtor.prototype[name] = baseCtor.prototype[nam
e];
        });
    });
}
```

理解这个例子

代码里首先定义了两个类，它们将做为mixins。 可以看到每个类都只定义了一个特定的行为或功能。 稍后我们使用它们来创建一个新类，同时具有这两种功能。

```
// Disposable Mixin
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }

}

// Activatable Mixin
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}
```

下面创建一个类，结合了这两个mixins。 下面来看一下具体是怎么操作的：

```
class SmartObject implements Disposable, Activatable {
```

首先应该注意到的是，没使用 `extends` 而是使用 `implements` 。 把类当成了接口，仅使用Disposable和Activatable的类型而非其实现。 这意味着我们需要在类里面实现接口。 但是这是我们在用mixin时想避免的。

我们可以这么做来达到目的，为将要mixin进来的属性方法创建出占位属性。 这告诉编译器这些成员在运行时是可用的。 这样就能使用mixin带来的便利，虽说需要提前定义一些占位属性。

```
// Disposable
isDisposed: boolean = false;
dispose: () => void;
// Activatable
isActive: boolean = false;
activate: () => void;
deactivate: () => void;
```

最后，把mixins混入定义的类，完成全部实现部分。

```
applyMixins(SmartObject, [Disposable, Activatable]);
```

最后，创建这个帮助函数，帮我们做混入操作。 它会遍历mixins上的所有属性，并复制到目标上去，把之前的占位属性替换成真正的实现代码。

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(n
ame => {
            derivedCtor.prototype[name] = baseCtor.prototype[nam
e];
        })
    });
}
```

三斜线指令是包含单个XML标签的单行注释。 注释的内容会做为编译器指令使用。

三斜线指令仅可放在包含它的文件的最顶端。 一个三斜线指令的前面只能出现单行或多行注释，这包括其它的三斜线指令。 如果它们出现在一个语句或声明之后，那么它们会被当做普通的单行注释，并且不具有特殊的涵义。

`/// <reference path="..." />`

`/// <reference path="..." />` 指令是三斜线指令中最常见的一种。 它用于声明文件间的依赖。

三斜线引用告诉编译器在编译过程中要引入的额外的文件。

当使用 `--out` 或 `--outFile` 时，它也可以做为调整输出内容顺序的一种方法。 文件在输出文件内容中的位置与经过预处理后的输入顺序一致。

预处理输入文件

编译器会对输入文件进行预处理来解析所有三斜线引用指令。 在这个过程中，额外的文件会加到编译过程中。

这个过程会以一些根文件开始； 它们是在命令行中指定的文件或是在 `tsconfig.json` 中的 `"files"` 列表里的文件。 这些根文件按指定的顺序进行预处理。 在一个文件被加入列表前，它包含的所有三斜线引用都要被处理，还有它们包含的目标。 三斜线引用以它们在文件里出现的顺序，使用深度优先的方式解析。

一个三斜线引用路径是相对于包含它的文件的，如果不是根文件。

错误

引用不存在的文件会报错。 一个文件用三斜线指令引用自己会报错。

使用 `--noResolve`

如果指定了 `--noResolve` 编译选项，三斜线引用会被忽略；它们不会增加新文件，也不会改变给定文件的顺序。

`/// <reference types="..." />`

与 `/// <reference path="..." />` 指令相似，这个指令是用来声明依赖的；一个 `/// <reference types="..." />` 指令则声明了对某个包的依赖。

对这些包的名字的解析与在 `import` 语句里对模块名的解析类似。 可以简单地把三斜线类型引用指令当做 `import` 声明的包。

例如，把 `/// <reference types="node" />` 引入到声明文件，表明这个文件使用了 `@types/node/index.d.ts` 里面声明的名字；并且，这个包需要在编译阶段与声明文件一起被包含进来。

仅当在你需要写一个 `d.ts` 文件时才使用这个指令。

对于那些在编译阶段生成的声明文件，编译器会自动地添加 `/// <reference types="..." />`；当且仅当结果文件中使用了引用的包里的声明时才会在生成的声明文件里添加 `/// <reference types="..." />` 语句。

若要在 `.ts` 文件里声明一个对 `@types` 包的依赖，使用 `--types` 命令行选项或在 `tsconfig.json` 里指定。 查看在 `tsconfig.json` 里使用 `@types`，`typeRoots` 和 `types` 了解详情。

`/// <reference no-default-lib="true"/>`

这个指令把一个文件标记成默认库。 你会在 `lib.d.ts` 文件和它不同的变体的顶端看到这个注释。

这个指令告诉编译器在编译过程中不要包含这个默认库（比如，`lib.d.ts`）。 这与在命令行上使用 `--noLib` 相似。

还要注意，当传递了 `--skipDefaultLibCheck` 时，编译器只会忽略检查带有 `/// <reference no-default-lib="true"/>` 的文件。

`/// <amd-module />`

默认情况下生成的AMD模块都是匿名的。 但是，当一些工具需要处理生成的模块时会产生问题，比如 `r.js`。

`amd-module` 指令允许给编译器传入一个可选的模块名：

amdModule.ts

```
///<amd-module name='NamedModule'/>
export class C {
}
```

这会将 `NamedModule` 传入到AMD `define` 函数里：

amdModule.js

```
define("NamedModule", ["require", "exports"], function (require,
 exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

`/// <amd-dependency />`

> 注意：这个指令被废弃了。使用 `import "moduleName";` 语句代替。

`/// <amd-dependency path="x" />` 告诉编译器有一个非TypeScript模块依赖需要被注入，做为目标模块 `require` 调用的一部分。

`amd-dependency` 指令也可以带一个可选的 `name` 属性；它允许我们为amd-dependency传入一个可选名字：

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

生成的JavaScript代码：

```
define(["require", "exports", "legacy/moduleA"], function (require, exports, moduleA) {
    moduleA.callStuff()
});
```

TypeScript 2.3以后的版本支持使用 `--checkJs` 对 `.js` 文件进行类型检查并提示错误的模式。

你可以通过添加 `// @ts-nocheck` 注释来忽略类型检查；相反你可以通过去掉 `--checkJs` 设置并添加 `// @ts-check` 注释来选则检查某些 `.js` 文件。你还可以使用 `// @ts-ignore` 来忽略本行的错误。

下面是一些值得注意的类型检查在 `.js` 文件与 `.ts` 文件上的差异：

在JSDoc上使用类型

`.js` 文件里，类型可以和在 `.ts` 文件里一样被推断出来。同样地，当类型不能被推断时，它们可以通过JSDoc来指定，就好比在 `.ts` 文件里那样。

JSDoc注解修饰的声明会被设置为这个声明的类型。比如：

```
/** @type {number} */
var x;

x = 0;      // OK
x = false;  // Error: boolean is not assignable to number
```

你可以在这里找到所有JSDoc支持的模式，JSDoc文档。

从类内部赋值语句推断属性声明

ES2015/ES6不存在类属性的声明。属性是动态的赋予的，就如同对象字面量一样。

在 `.js` 文件里，属性声明是由类内部的属性赋值语句推断出来的。属性的类型是赋值语句右侧所有值的联合。构造函数里定义的属性是永远存在的，在方法存取器里定义的被认为是可选的。

使用JSDoc修饰属性赋值来指定属性类型。例如：

```
class C {
  constructor() {
    /** @type {number | undefined} */
    this.prop = undefined;

  }
}


let c = new C();
c.prop = 0;          // OK
c.prop = "string";  // Error: string is not assignable to number
|undefined
```

如果属性永远都不在类的内部被设置，那么它们被当成是未知的。如果类具有只读的属性，考虑在构造函数里给它初始化成 `undefined` ，例如 `this.prop = undefined;` 。

CommonJS模块输入支持

`.js` 文件支持将CommonJS模块做为输入模块格式。
对 `exports` 和 `module.exports` 的赋值被识别为导出声明。 相似地， `require` 函数调用被识别为模块导入。例如：

```
// import module "fs"
const fs = require("fs");


// export function readFile
module.exports.readFile = function(f) {
  return fs.readFileSync(f);
}
```

对象字面量是开放的

默认地，变量声明中的对象字面量本身就提供了类型声明。新的成员不能被加到对象中去。这个规则在 `.js` 文件里被放宽了；对象字面量具有开放的类型，允许添加并访问原先没有定义的属性。例如：

```
var obj = { a: 1 };
obj.b = 2;  // Allowed
```

对象字面量具有默认的索引签名 `[x:string]: any` ，它们可以被当成开放的映射而不是封闭的对象。

与其它JS检查行为相似，这种行为可以通过指定JSDoc类型来改变，例如：

```
/** @type {{a: number}} */
var obj = { a: 1 };
obj.b = 2;  // Error, type {a: number} does not have property b
```

函数参数是默认可选的

由于JS不支持指定可选参数（不指定一个默认值）， `.js` 文件里所有函数参数都被当做可选的。使用比预期少的参数调用函数是允许的。

需要注意的一点是，使用过多的参数调用函数会得到一个错误。

例如：

```
function bar(a, b){
  console.log(a + " " + b);
}

bar(1);       // OK, second argument considered optional
bar(1, 2);
bar(1, 2, 3); // Error, too many arguments
```

使用JSDoc注解的函数会被从这条规则里移除。使用JSDoc可选参数语法来表示可选性。比如：

```
/**
 * @param {string} [somebody] - Somebody's name.
 */
function sayHello(somebody) {
    if (!somebody) {
        somebody = 'John Doe';
    }
    alert('Hello ' + somebody);
}

sayHello();
```

由 `arguments` 推断出的var-args参数声明

如果一个函数的函数体内有对 `arguments` 的引用，那么这个函数会隐式地被认为具有一个var-arg参数（比如: `(...arg: any[]) => any`)）。使用JSDoc的var-arg语法来指定 `arguments` 的类型。

未指定的类型参数默认为 `any`

未指定的泛型参数类型将默认为 `any` 。有如下几种情形：

在extends语句中

例如， `React.Component` 被定义成具有两个泛型参数， `Props` 和 `State` 。 在一个 `.js` 文件里，没有一个合法的方式在extends语句里指定它们。默认地参数类型为 `any` ：

```
import { Component } from "react";

class MyComponent extends Component {
    render() {
        this.props.b; // Allowed, since this.props is of type any

    }
}
```

使用JSDoc的 `@augments` 来明确地指定类型。例如：

```
import { Component } from "react";

/**
 * @augments {Component<{a: number}, State>}
 */
class MyComponent extends Component {
    render() {
        this.props.b; // Error: b does not exist on {a:number}
    }
}
```

在JSDoc引用中

JSDoc里未指定的泛型参数默认为 `any` ：

```
/** @type{Array} */
var x = [];

x.push(1);        // OK
x.push("string"); // OK, x is of type Array<any>


/** @type{Array.<number>} */
var y = [];

y.push(1);        // OK
y.push("string"); // Error, string is not assignable to number
```

在函数调用中

泛型函数的调用使用 `arguments` 来推断泛型参数。有时候，这个流程不能够推断出类型，大多是因为缺少推断的源；在这种情况下，泛型参数类型默认为 `any` 。例如：

```
var p = new Promise((resolve, reject) => { reject() });

p; // Promise<any>;
```

这篇指南的目的是教你如何书写高质量的TypeScript声明文件。

在这篇指南里，我们假设你对TypeScript已经有了基本的了解。 如果没有，请先阅读TypeScript手册 来了解一些基本知识，尤其是类型和命名空间部分。

章节

这篇指南被分成了以下章节。

结构

结构一节将帮助你了解常见库的格式以及如何为每种格式书写正确的声明文件。 如果你在编辑一个已经存在的文件，那么你可能不需要阅读此章节。 如果你在书写新的声明文件，那么你必须阅读此章节以理解库的不同格式是如何影响声明文件的书写的。

举例

很多时候，我们只能通过一些示例来了解第三方库是如何工作的，同时我们需要为这样的库书写声明文件。 举例一节展示了很多常见的API模式以及如何为它们书写声明文件。 这篇指南是针对TypeScript初学者的，它们可能还不了解TypeScript里的所有语言结构。

规范

声明文件里有很多常见的错误是很容易避免的。 规范一节指出了常见的错误， 描述了如何发现它们， 与怎样去修复。 每个人都要阅读这个章节以了解如何避免常见错误。

深入

对于那些对声明文件底层工作机制感兴趣的老手们， 深入一节解释了很多高级书写声明文件的高级概念， 以及展示了如何利用这些概念来创建整洁和直观的声明文件。

模版

在模版一节里，你能找到一些声明文件，它们可以帮助你快速开始 当你在书写一个新声明文件的时候。 参考结构这篇文档来找到应该使用哪个模版文件。

发布到npm

发布一节讲解了如何发布声明文件为npm包，及如何管理包的依赖。

查找与安装声明文件

对于JavaScript库的使用者来讲，使用一节提供了一些简单步骤来定位与安装相应的声明文件。

概述

一般来讲，你组织声明文件的方式取决于库是如何被使用的。 在JavaScript中一个库有很多使用方式，这就需要你书写声明文件去匹配它们。 这篇指南涵盖了如何识别常见库的模式，和怎样书写符合相应模式的声明文件。

针对每种主要的库的组织模式，在模版一节都有对应的文件。 你可以利用它们帮助你快速上手。

识别库的类型

首先，我们先看一下TypeScript声明文件能够表示的库的类型。 这里会简单展示每种类型的库的使用方式，如何去书写，还有一些真实案例。

识别库的类型是书写声明文件的第一步。 我们将会给出一些提示，关于怎样通过库的使用方法及其源码来识别库的类型。 根据库的文档及组织结构不同，这两种方式可能一个会比另外的那个简单一些。 我们推荐你使用任意你喜欢的方式。

全局库

全局库是指能在全局命名空间下访问的（例如：不需要使用任何形式的 `import` ）。 许多库都是简单的暴露出一个或多个全局变量。 比如，如果你使用过jQuery， `$` 变量可以被够简单的引用：

```
$(() => { console.log('hello!'); } );
```

你经常会在全局库的指南文档上看到如何在HTML里用脚本标签引用库：

```
<script src="http://a.great.cdn.for/someLib.js"></script>
```

目前，大多数流行的全局访问型库实际上都以UMD库的形式进行书写（见后文）。 UMD库的文档很难与全局库文档两者之间难以区分。 在书写全局声明文件前，一定要确认一下库是否真的不是UMD。

从代码上识别全局库

全局库的代码通常都十分简单。 一个全局的“Hello, world”库可能是这样的：

```
function createGreeting(s) {
    return "Hello, " + s;
}
```

或这样：

```
window.createGreeting = function(s) {
    return "Hello, " + s;
}
```

当你查看全局库的源代码时，你通常会看到：

- 顶级的 `var` 语句或 `function` 声明
- 一个或多个赋值语句到 `window.someName`
- 假设DOM原始值像 `document` 或 `window` 是存在的

你不会看到：

- 检查是否使用或如何使用模块加载器，比如 `require` 或 `define`
- CommonJS/Node.js风格的导入如 `var fs = require("fs");`
- `define(...)` 调用
- 文档里说明了如何去 `require` 或导入这个库

全局库的例子

由于把一个全局库转变成UMD库是非常容易的，所以很少流行的库还再使用全局的风格。 然而，小型的且需要DOM（或没有依赖）的库可能还是全局类型的。

全局库模版

模版文件 `global.d.ts` 定义了 `myLib` 库作为例子。 一定要阅读"防止命名冲突"补充说明。

模块化库

一些库只能工作在模块加载器的环境下。 比如，像 `express` 只能在Node.js里工作所以必须使用CommonJS的 `require` 函数加载。

ECMAScript 2015（也就是ES2015，ECMAScript 6或ES6），CommonJS和RequireJS具有相似的导入一个模块的表示方法。 例如，对于JavaScript CommonJS （Node.js），有下面的代码

```
var fs = require("fs");
```

对于TypeScript或ES6， `import` 关键字也具有相同的作用：

```
import fs = require("fs");
```

你通常会在模块化库的文档里看到如下说明：

```
var someLib = require('someLib');
```

或

```
define(..., ['someLib'], function(someLib) {

});
```

与全局模块一样，你也可能会在UMD模块的文档里看到这些例子，因此要仔细查看源码和文档。

从代码上识别模块化库

模块库至少会包含下列具有代表性的条目之一：

- 无条件的调用 `require` 或 `define`
- 像 `import * as a from 'b';` or `export c;` 这样的声明
- 赋值给 `exports` 或 `module.exports`

它们极少包含：

- 对 `window` 或 `global` 的赋值

模块化库的例子

许多流行的Node.js库都是这种模块化的，例如 `express` ， `gulp` 和 `request` 。

UMD

*UMD*模块是指那些既可以作为模块使用（通过导入）又可以作为全局（在没有模块加载器的环境里）使用的模块。 许多流行的库，比如Moment.js，就是这样的形式。 比如，在Node.js或RequireJS里，你可以这样写：

```
import moment = require("moment");
console.log(moment.format());
```

然而在纯净的浏览器环境里你也可以这样写：

```
console.log(moment.format());
```

识别UMD库

UMD模块会检查是否存在模块加载器环境。 这是非常形容观察到的模块，它们会像下面这样：

```
(function (root, factory) {
    if (typeof define === "function" && define.amd) {
        define(["libName"], factory);
    } else if (typeof module === "object" && module.exports) {
        module.exports = factory(require("libName"));
    } else {
        root.returnExports = factory(root.libName);
    }
}(this, function (b) {
```

如果你在库的源码里看到了 `typeof define` ， `typeof window` ，或 `typeof module` 这样的测试，尤其是在文件的顶端，那么它几乎就是一个UMD库。

UMD库的文档里经常会包含通过 `require` “在Node.js里使用”例子， 和“在浏览器里使用”的例子，展示如何使用 `<script>` 标签去加载脚本。

UMD库的例子

大多数流行的库现在都能够被当成UMD包。 比如jQuery,Moment.js,lodash和许多其它的。

模版

针对模块有三种可用的模块， `module.d.ts` , `module-class.d.ts` and `module-function.d.ts` .

使用 `module-function.d.ts` ，如果模块能够作为函数调用。

```
var x = require("foo");
// Note: calling 'x' as a function
var y = x(42);
```

一定要阅读补充说明： “ES6模块调用签名的影响”

使用 `module-class.d.ts` 如果模块能够使用 `new` 来构造：

```
var x = require("bar");
// Note: using 'new' operator on the imported variable
var y = new x("hello");
```

相同的补充说明作用于这些模块。

如果模块不能被调用或构造，使用 `module.d.ts` 文件。

模块插件或*UMD*插件

一个模块插件可以改变一个模块的结构（UMD或模块）。例如，在Moment.js里，`moment-range`添加了新的`range`方法到`monent`对象。

对于声明文件的目标，我们会写相同的代码不论被改变的模块是一个纯粹的模块还是UMD模块。

模版

使用`module-plugin.d.ts`模版。

全局插件

一个全局插件是全局代码，它们会改变全局对象的结构。对于全局修改的模块，在运行时存在冲突的可能。

比如，一些库往`Array.prototype`或`String.prototype`里添加新的方法。

识别全局插件

全局通常很容易地从它们的文档识别出来。

你会看到像下面这样的例子：

```
var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

模版

使用`global-plugin.d.ts`模版。

全局修改的模块

当一个全局修改的模块被导入的时候，它们会改变全局作用域里的值。 比如，存在一些库它们添加新的成员到 `String.prototype` 当导入它们的时候。 这种模式很危险，因为可能造成运行时的冲突， 但是我们仍然可以为它们书写声明文件。

识别全局修改的模块

全局修改的模块通常可以很容易地从它们的文档识别出来。 通常来讲，它们与全局插件相似，但是需要 `require` 调用来激活它们的效果。

你可能会看到像下面这样的文档:

```
// 'require' call that doesn't use its return value
var unused = require("magic-string-time");
/* or */
require("magic-string-time");

var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

模版

使用 `global-modifying-module.d.ts` 模版。

使用依赖

可能会有以下几种依赖。

依赖全局库

如果你的库依赖于某个全局库，使用 `/// <reference types="..." />` 指令：

```
/// <reference types="someLib" />

function getThing(): someLib.thing;
```

依赖模块

如果你的库依赖于模块，使用 `import` 语句：

```
import * as moment from "moment";

function getThing(): moment;
```

依赖UMD库

从全局库

如果你的全局库依赖于某个UMD模块，使用 `/// <reference types` 指令：

```
/// <reference types="moment" />

function getThing(): moment;
```

从一个模块或UMD库

如果你的模块或UMD库依赖于一个UMD库，使用 `import` 语句：

```
import * as someLib from 'someLib';
```

不要使用 `/// <reference` 指令去声明UMD库的依赖！

补充说明

防止命名冲突

注意，在书写全局声明文件时，允许在全局作用域里定义很多类型。 我们十分不建义这样做，当一个工程里有许多声明文件时，它会导致无法处理的命名冲突。

一个简单的规则是使用库定义的全局变量名来声明命名空间类型。 比如，库定义了一个全局的值 `cats` ，你可以这样写

```
declare namespace cats {
    interface KittySettings { }
}
```

不要

```
// at top-level
interface CatsKittySettings { }
```

这样也保证了库在转换成UMD的时候没有任何的破坏式改变，对于声明文件用户来说。

ES6模块插件的影响

一些插件添加或修改已存在的顶层模块的导出部分。 当然这在CommonJS和其它加载器里是允许的，ES模块被当作是不可改变的因此这种模式就不可行了。 因为TypeScript是能不预知加载器类型的，所以没没在编译时保证，但是开发者如果要转到ES6模块加载器上应该注意这一点。

ES6模块调用签名的影响

很多流行库，比如Express，暴露出自己作为可以调用的函数。 比如，典型的Express使用方法如下：

```
import exp = require("express");
var app = exp();
```

在ES6模块加载器里，顶层的对象（这里以 `exp` 导入）只能具有属性；顶层的模块对象永远不能被调用。 十分常见的解决方法是定义一个 `default` 导出到一个可调用的/可构造的对象；一会模块加载器助手工具能够自己探测到这种情况并且使用 `default` 导出来替换顶层对象。

普通类型

`Number` ， `String` ， `Boolean` 和 `Object`

不要使用如下类型 `Number` ， `String` ， `Boolean` 或 `Object` 。 这些类型指的是非原始的装盒对象，它们几乎没在JavaScript代码里正确地使用过。

```
/* 错误 */
function reverse(s: String): String;
```

应该使用类型 `number` ， `string` ，and `boolean` 。

```
/* OK */
function reverse(s: string): string;
```

使用非原始的 `object` 类型来代替 `Object` （TypeScript 2.2新增）

泛型

不要定义一个从来没使用过其类型参数的泛型类型。 了解详情TypeScript FAQ page。

回调函数类型

回调函数返回值类型

不要为返回值被忽略的回调函数设置一个 `any` 类型的返回值类型：

```
/* 错误 */
function fn(x: () => any) {
    x();
}
```

应该给返回值被忽略的回调函数设置 `void` 类型的返回值类型：

```
/* OK */
function fn(x: () => void) {
    x();
}
```

为什么：使用 `void` 相对安全，因为它防止了你不小心使用 `x` 的返回值：

```
function fn(x: () => void) {
    var k = x(); // oops! meant to do something else
    k.doSomething(); // error, but would be OK if the return typ
e had been 'any'
}
```

回调函数里的可选参数

不要在回调函数里使用可选参数除非你真的要这么做：

```
/* 错误 */
interface Fetcher {
    getObject(done: (data: any, elapsedTime?: number) => void):
void;
}
```

这里有一种特殊的意义：`done` 回调函数可能以1个参数或2个参数调用。 代码大概的意思是说这个回调函数不在乎是否有 `elapsedTime` 参数， 但是不需要把这个参数当成可选参数来达到此目的 -- 因为总是允许提供一个接收较少参数的回调函数。

应该写出回调函数的非可选参数：

```
/* OK */
interface Fetcher {
    getObject(done: (data: any, elapsedTime: number) => void): v
oid;
}
```

重载与回调函数

不要因为回调函数参数个数不同而写不同的重载：

```
/* 错误 */
declare function beforeAll(action: () => void, timeout?: number)
: void;
declare function beforeAll(action: (done: DoneFn) => void, timeo
ut?: number): void;
```

应该只使用最大参数个数写一个重载：

```
/* OK */
declare function beforeAll(action: (done: DoneFn) => void, timeo
ut?: number): void;
```

为什么：回调函数总是可以忽略某个参数的，因此没必要为参数少的情况写重载。参数少的回调函数首先允许错误类型的函数被传入，因为它们匹配第一个重载。

函数重载

顺序

不要把一般的重载放在精确的重载前面：

```
/* 错误 */
declare function fn(x: any): any;
declare function fn(x: HTMLElement): number;
declare function fn(x: HTMLDivElement): string;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: any, wat?
```

应该排序重载令精确的排在一般的之前：

```
/* OK */
declare function fn(x: HTMLDivElement): string;
declare function fn(x: HTMLElement): number;
declare function fn(x: any): any;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: string, :)
```

为什么：TypeScript会选择第一个匹配到的重载当解析函数调用的时候。当前面的重载比后面的“普通”，那么后面的被隐藏了不会被调用。

使用可选参数

不要为仅在末尾参数不同时写不同的重载：

```
/* 错误 */
interface Example {
    diff(one: string): number;
    diff(one: string, two: string): number;
    diff(one: string, two: string, three: boolean): number;
}
```

应该尽可能使用可选参数：

```
/* OK */
interface Example {
    diff(one: string, two?: string, three?: boolean): number;
}
```

注意这在所有重载都有相同类型的返回值时会不好用。

为什么：有以下两个重要原因。

TypeScript解析签名兼容性时会查看是否某个目标签名能够使用源的参数调用，且允许外来参数。下面的代码暴露出一个bug，当签名被正确的使用可选参数书写时：

```
function fn(x: (a: string, b: number, c: number) => void) { }
var x: Example;
// When written with overloads, OK -- used first overload
// When written with optionals, correctly an error
fn(x.diff);
```

第二个原因是当使用了TypeScript"严格检查null"特性时。因为没有指定的参数在JavaScript里表示为 `undefined` ，通常显示地为可选参数传入一个 `undefined` 。这段代码在严格null模式下可以工作：

```
var x: Example;
// When written with overloads, incorrectly an error because of
passing 'undefined' to 'string'
// When written with optionals, correctly OK
x.diff("something", true ? undefined : "hour");
```

使用联合类型

不要为仅在某个位置上的参数类型不同的情况下定义重载：

```
/* WRONG */
interface Moment {
    utcOffset(): number;
    utcOffset(b: number): Moment;
    utcOffset(b: string): Moment;
}
```

应该尽可能地使用类型类型：

```
/* OK */
interface Moment {
    utcOffset(): number;
    utcOffset(b: number|string): Moment;
}
```

注意我们没有让 `b` 成为可选的，因为签名的返回值类型不同。

为什么：This is important for people who are "passing through" a value to your function:

```
function fn(x: string): void;
function fn(x: number): void;
function fn(x: number|string) {
    // When written with separate overloads, incorrectly an error

    // When written with union types, correctly OK
    return moment().utcOffset(x);
}
```

简介

这篇指南的目的是教你如何书写高质量的TypeScript声明文件。 我们在这里会展示一些API的文档，还有它们的使用示例， 并且阐述了如何为它们书写声明文件。

这些例子是按复杂度递增的顺序组织的。

- 全局变量
- 全局函数
- 带属性的对象
- 函数重载
- 可重用类型（接口）
- 可重用类型（类型别名）
- 组织类型
- 类

例子

全局变量

文档

> 全局变量 `foo` 包含了存在组件总数。

代码

```
console.log("Half the number of widgets is " + (foo / 2));
```

声明

使用 `declare var` 声明变量。 如果变量是只读的，那么可以使用 `declare const` 。 你还可以使用 `declare let` 如果变量拥有块级作用域。

```
/** 组件总数 */
declare var foo: number;
```

全局函数

文档

> 用一个字符串参数调用 `greet` 函数向用户显示一条欢迎信息。

代码

```
greet("hello, world");
```

声明

使用 `declare function` 声明函数。

```
declare function greet(greeting: string): void;
```

带属性的对象

文档

> 全局变量 `myLib` 包含一个 `makeGreeting` 函数，还有一个属性 `numberOfGreetings` 指示目前为止欢迎数量。

代码

```
let result = myLib.makeGreeting("hello, world");
console.log("The computed greeting is:" + result);

let count = myLib.numberOfGreetings;
```

声明

使用 `declare namespace` 描述用点表示法访问的类型或值。

```
declare namespace myLib {
    function makeGreeting(s: string): string;
    let numberOfGreetings: number;
}
```

函数重载

文档

> `getWidget` 函数接收一个数字，返回一个组件，或接收一个字符串并返回一个组件数组。

代码

```
let x: Widget = getWidget(43);

let arr: Widget[] = getWidget("all of them");
```

声明

```
declare function getWidget(n: number): Widget;
declare function getWidget(s: string): Widget[];
```

可重用类型（接口）

文档

> 当指定一个欢迎词时，你必须传入一个 `GreetingSettings` 对象。 这个对象具有以下几个属性：
>
> 1- greeting：必需的字符串
>
> 2- duration: 可靠的时长（毫秒表示）
>
> 3- color: 可选字符串，比如‘#ff00ff’

代码

```
greet({
  greeting: "hello world",
  duration: 4000
});
```

声明

使用 `interface` 定义一个带有属性的类型。

```
interface GreetingSettings {
  greeting: string;
  duration?: number;
  color?: string;
}

declare function greet(setting: GreetingSettings): void;
```

可重用类型（类型别名）

文档

> 在任何需要欢迎词的地方，你可以提供一个 `string` ，一个返回 `string` 的函数或一个 `Greeter` 实例。

代码

```
function getGreeting() {
    return "howdy";
}
class MyGreeter extends Greeter { }

greet("hello");
greet(getGreeting);
greet(new MyGreeter());
```

声明

你可以使用类型别名来定义类型的短名：

```
type GreetingLike = string | (() => string) | MyGreeter;

declare function greet(g: GreetingLike): void;
```

组织类型

文档

> `greeter` 对象能够记录到文件或显示一个警告。你可以为 `.log(...)` 提供LogOptions和为 `.alert(...)` 提供选项。

代码

```
const g = new Greeter("Hello");
g.log({ verbose: true });
g.alert({ modal: false, title: "Current Greeting" });
```

声明

使用命名空间组织类型。

```
declare namespace GreetingLib {
    interface LogOptions {
        verbose?: boolean;
    }
    interface AlertOptions {
        modal: boolean;
        title?: string;
        color?: string;
    }
}
```

你也可以在一个声明中创建嵌套的命名空间：

```
declare namespace GreetingLib.Options {
    // Refer to via GreetingLib.Options.Log
    interface Log {
        verbose?: boolean;
    }
    interface Alert {
        modal: boolean;
        title?: string;
        color?: string;
    }
}
```

类

文档

> 你可以通过实例化 `Greeter` 对象来创建欢迎词，或者继承 `Greeter` 对象来自定义欢迎词。

代码

```
const myGreeter = new Greeter("hello, world");
myGreeter.greeting = "howdy";
myGreeter.showGreeting();

class SpecialGreeter extends Greeter {
    constructor() {
        super("Very special greetings");
    }
}
```

声明

使用 `declare class` 描述一个类或像类一样的对象。 类可以有属性和方法，就和构造函数一样。

```
declare class Greeter {
    constructor(greeting: string);

    greeting: string;
    showGreeting(): void;
}
```

声明文件原理：深入探究

组织模块以提供你想要的API形式保持一致是比较难的。 比如，你可能想要这样一个模块，可以用或不用 `new` 来创建不同的类型， 在不同层级上暴露出不同的命名类型， 且模块对象上还带有一些属性。

阅读这篇指定后，你就会了解如果书写复杂的暴露出友好API的声明文件。 这篇指定针对于模块（UMD）库，因为它们的选择具有更高的可变性。

核心概念

如果你理解了一些关于TypeScript是如何工作的核心概念， 那么你就能够为任何结构书写声明文件。

类型

如果你正在阅读这篇指南，你可能已经大概了解TypeScript里的类型指是什么。 明确一下，类型通过以下方式引入：

- 类型别名声明（ `type sn = number | string;` ）
- 接口声明（ `interface I { x: number[]; }` ）
- 类声明（ `class C { }` ）
- 枚举声明（ `enum E { A, B, C }` ）
- 指向某个类型的 `import` 声明

以上每种声明形式都会创建一个新的类型名称。

值

与类型相比，你可能已经理解了什么是值。 值是运行时名字，可以在表达式里引用。 比如 `let x = 5;` 创建一个名为 `x` 的值。

同样，以下方式能够创建值：

- `let` ， `const` ，和 `var` 声明
- 包含值的 `namespace` 或 `module` 声明

- `enum` 声明
- `class` 声明
- 指向值的 `import` 声明
- `function` 声明

命名空间

类型可以存在于命名空间里。 比如，有这样的声明 `let x: A.B.C` ， 我们就认为 `C` 类型来自 `A.B` 命名空间。

这个区别虽细微但很重要 -- 这里， `A.B` 不是必需的类型或值。

简单的组合：一个名字，多种意义

一个给定的名字 `A` ，我们可以找出三种不同的意义：一个类型，一个值或一个命名空间。 要如何去解析这个名字要看它所在的上下文是怎样的。 比如，在声明 `let m: A.A = A;` ， `A` 首先被当做命名空间，然后做为类型名，最后是值。 这些意义最终可能会指向完全不同的声明！

这看上去另人迷惑，但是只要我们不过度的重载这还是很方便的。 下面让我们来看看一些有用的组合行为。

内置组合

眼尖的读者可能会注意到，比如， `class` 同时出现在类型和值列表里。 `class C { }` 声明创建了两个东西： 类型 `C` 指向类的实例结构， 值 `C` 指向类构造函数。 枚举声明拥有相似的行为。

用户组合

假设我们写了模块文件 `foo.d.ts` ：

```
export var SomeVar: { a: SomeType };
export interface SomeType {
  count: number;
}
```

这样使用它：

```
import * as foo from './foo';
let x: foo.SomeType = foo.SomeVar.a;
console.log(x.count);
```

这可以很好地工作，但是我们知道 `SomeType` 和 `SomeVar` 很相关 因此我们想让他们有相同的名字。 我们可以使用组合通过相同的名字 `Bar` 表示这两种不同的对象（值和对象）：

```
export var Bar: { a: Bar };
export interface Bar {
  count: number;
}
```

这提供了解构使用的机会：

```
import { Bar } from './foo';
let x: Bar = Bar.a;
console.log(x.count);
```

再次地，这里我们使用 `Bar` 做为类型和值。 注意我们没有声明 `Bar` 值为 `Bar` 类型 -- 它们是独立的。

高级组合

有一些声明能够通过多个声明组合。 比如， `class C { }` 和 `interface C { }` 可以同时存在并且都可以做为 `C` 类型的属性。

只要不产生冲突就是合法的。 一个普通的规则是值总是会和同名的其它值产生冲突除非它们在不同命名空间里， 类型冲突则发生在使用类型别名声明的情况下（ `type s = string` ）， 命名空间永远不会发生冲突。

让我们看看如何使用。

利用 `interface` 添加

我们可以使用一个 `interface` 往别一个 `interface` 声明里添加额外成员：

```
interface Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

这同样作用于类：

```
class Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

注意我们不能使用接口往类型别名里添加成员（ `type s = string;` ）

使用 `namespace` 添加

`namespace` 声明可以用来添加新类型，值和命名空间，只要不出现冲突。

比如，我们可能添加静态成员到一个类：

```
class C {
}
// ... elsewhere ...
namespace C {
  export let x: number;
}
let y = C.x; // OK
```

注意在这个例子里，我们添加一个值到 `C` 的静态部分（它的构造函数）。这里因为我们添加了一个值，且其它值的容器是另一个值（类型包含于命名空间，命名空间包含于另外的命名空间）。

我们还可以给类添加一个命名空间类型：

```
class C {
}
// ... elsewhere ...
namespace C {
  export interface D { }
}
let y: C.D; // OK
```

在这个例子里，直到我们写了 `namespace` 声明才有了命名空间 `C`。做为命名空间的 `C` 不会与类创建的值 `C` 或类型 `C` 相互冲突。

最后，我们可以进行不同的合并通过 `namespace` 声明。Finally, we could perform many different merges using `namespace` declarations. This isn't a particularly realistic example, but shows all sorts of interesting behavior:

```
namespace X {
  export interface Y { }
  export class Z { }
}

// ... elsewhere ...
namespace X {
  export var Y: number;
  export namespace Z {
    export class C { }
  }
}
type X = string;
```

在这个例子里，第一个代码块创建了以下名字与含义：

- 一个值 `X` （因为 `namespace` 声明包含一个值， `Z` ）
- 一个命名空间 `X` （因为 `namespace` 声明包含一个值， `Z` ）
- 在命名空间 `X` 里的类型 `Y`
- 在命名空间 `X` 里的类型 `Z` （类的实例结构）
- 值 `X` 的一个属性值 `Z` （类的构造函数）

第二个代码块创建了以下名字与含义：

- 值 `Y` （ `number` 类型），它是值 `X` 的一个属性
- 一个命名空间 `Z`
- 值 `Z` ，它是值 `X` 的一个属性
- 在 `X.Z` 命名空间下的类型 `C`
- 值 `X.Z` 的一个属性值 `C`
- 类型 `X`

使用 `export =` 或 `import`

一个重要的原则是 `export` 和 `import` 声明会导出或导入目标的所有含义。

现在我们已经按照指南里的步骤写好一个声明文件，是时候把它发布到npm了。 有两种主要方式用来发布声明文件到npm：

1. 与你的npm包捆绑在一起，或
2. 发布到npm上的@types organization。

如果你能控制要使用你发布的声明文件的那个npm包的话，推荐第一种方式。 这样的话，你的声明文件与JavaScript总是在一起传递。

包含声明文件到你的npm包

如果你的包有一个主 `.js` 文件，你还是需要在 `package.json` 里指定主声明文件。 设置 `types` 属性指向捆绑在一起的声明文件。 比如：

```
{
    "name": "awesome",
    "author": "Vandelay Industries",
    "version": "1.0.0",
    "main": "./lib/main.js",
    "types": "./lib/main.d.ts"
}
```

注意 `"typings"` 与 `"types"` 具有相同的意义，也可以使用它。

同样要注意的是如果主声明文件名是 `index.d.ts` 并且位置在包的根目录里（与 `index.js` 并列），你就不需要使用 `"types"` 属性指定了。

依赖

所有的依赖是由npm管理的。 确保所依赖的声明包都在 `package.json` 的 `"dependencies"` 里指明了 比如，假设我们写了一个包它依赖于Browserify和TypeScript。

```
{
    "name": "browserify-typescript-extension",
    "author": "Vandelay Industries",
    "version": "1.0.0",
    "main": "./lib/main.js",
    "types": "./lib/main.d.ts",
    "dependencies": {
        "browserify": "latest",
        "@types/browserify": "latest",
        "typescript": "next"
    }
}
```

这里，我们的包依赖于`browserify`和`typescript`包。`browserify`没有把它的声明文件捆绑在它的npm包里，所以我们需要依赖于`@types/browserify`得到它的声明文件。`typescript`相反，它把声明文件放在了npm包里，因此我们不需要依赖额外的包。

我们的包要从这两个包里暴露出声明文件，因此`browserify-typescript-extension`的用户也需要这些依赖。正因此，我们使用`"dependencies"`而不是`"devDependencies"`，否则用户将需要手动安装那些包。如果我们只是在写一个命令行应用，并且我们的包不会被当做一个库使用的话，那么我就可以使用`devDependencies`。

危险信号

`/// <reference path="..." />`

不要在声明文件里使用`/// <reference path="..." />`。

```
/// <reference path="../typescript/lib/typescriptServices.d.ts"
/>
....
```

应该使用`/// <reference types="..." />`代替

```
/// <reference types="typescript" />
....
```

务必阅读使用依赖一节了解详情。

打包所依赖的声明

如果你的类型声明依赖于另一个包：

- 不要把依赖的包放进你的包里，保持它们在各自的文件里。
- 不要将声明拷贝到你的包里。
- 应该依赖于npm类型声明包，如果依赖包没包含它自己的声明的话。

公布你的声明文件

在发布声明文件包之后，确保在DefinitelyTyped外部包列表里面添加一条引用。 这可以让查找工具知道你的包提供了自己的声明文件。

发布到@types

@types下面的包是从DefinitelyTyped里自动发布的，通过types-publisher工具。 如果想让你的包发布为@types包，提交一个pull request到https://github.com/DefinitelyTyped/DefinitelyTyped。 在这里查看详细信息contribution guidelines page。

在TypeScript 2.0，获取、使用和查找声明文件变得十分容易。 这篇文章将详细说明怎么做这三件事。

下载

在TypeScript 2.0以上的版本，获取类型声明文件只需要使用npm。

比如，获取lodash库的声明文件，只需使用下面的命令：

```
npm install --save @types/lodash
```

如果一个npm包像Publishing里所讲的一样已经包含了它的声明文件，那就不必再去下载相应的 `@types` 包了。

使用

下载完后，就可以直接在TypeScript里使用lodash了。 不论是在模块里还是全局代码里使用。

比如，你已经 `npm install` 安装了类型声明，你可以使用导入：

```
import * as _ from "lodash";
_.padStart("Hello TypeScript!", 20, " ");
```

或者如果你没有使用模块，那么你只需使用全局的变量 `_` 。

```
_.padStart("Hello TypeScript!", 20, " ");
```

查找

大多数情况下，类型声明包的名字总是与它们在 `npm` 上的包的名字相同，但是有 `@types/` 前缀， 但如果你需要的话，你可以在https://aka.ms/types这里查找你喜欢的库。

> 注意：如果你要找的声明文件不存在，你可以贡献一份，这样就方便了下一位要使用它的人。 查看DefinitelyTyped贡献指南页了解详情。

概述

如果一个目录下存在一个 `tsconfig.json` 文件，那么它意味着这个目录是TypeScript项目的根目录。 `tsconfig.json` 文件中指定了用来编译这个项目的根文件和编译选项。 一个项目可以通过以下方式之一来编译：

使用tsconfig.json

- 不带任何输入文件的情况下调用 `tsc` ，编译器会从当前目录开始去查找 `tsconfig.json` 文件，逐级向上搜索父目录。
- 不带任何输入文件的情况下调用 `tsc` ，且使用命令行参数 `--project` （或 `-p` ）指定一个包含 `tsconfig.json` 文件的目录。

当命令行上指定了输入文件时， `tsconfig.json` 文件会被忽略。

示例

`tsconfig.json` 示例文件:

- 使用 `"files"` 属性

```
{
    "compilerOptions": {
        "module": "commonjs",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    },
    "files": [
        "core.ts",
        "sys.ts",
        "types.ts",
        "scanner.ts",
        "parser.ts",
        "utilities.ts",
        "binder.ts",
        "checker.ts",
        "emitter.ts",
        "program.ts",
        "commandLineParser.ts",
        "tsc.ts",
        "diagnosticInformationMap.generated.ts"
    ]
}
```

- 使用 `"include"` 和 `"exclude"` 属性

```
{
    "compilerOptions": {
        "module": "system",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "outFile": "../../built/local/tsc.js",
        "sourceMap": true
    },
    "include": [
        "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}
```

细节

`"compilerOptions"` 可以被忽略，这时编译器会使用默认值。在这里查看完整的编译器选项列表。

`"files"` 指定一个包含相对或绝对文件路径的列表。
`"include"` 和 `"exclude"` 属性指定一个文件glob匹配模式列表。支持的glob通配符有：

- `*` 匹配0或多个字符（不包括目录分隔符）
- `?` 匹配一个任意字符（不包括目录分隔符）
- `**/` 递归匹配任意子目录

如果一个glob模式里的某部分只包含 `*` 或 `.*` ，那么仅有支持的文件扩展名类型被包含在内（比如默认 `.ts` ， `.tsx` ，和 `.d.ts` ，如果 `allowJs` 设置能 `true` 还包含 `.js` 和 `.jsx` ）。

如果 `"files"` 和 `"include"` 都没有被指定，编译器默认包含当前目录和子目录下所有的TypeScript文件（ `.ts` , `.d.ts` 和 `.tsx` ），排除在 `"exclude"` 里指定的文件。JS文件（ `.js` 和 `.jsx` ）也被包含进来如果 `allowJs` 被设置

成 `true` 。 如果指定了 `"files"` 或 `"include"` ，编译器会将它们结合一并包含进来。 使用 `"outDir"` 指定的目录下的文件永远会被编译器排除，除非你明确地使用 `"files"` 将其包含进来（这时就算用 `exclude` 指定也没用）。

使用 `"include"` 引入的文件可以使用 `"exclude"` 属性过滤。 然而，通过 `"files"` 属性明确指定的文件却总是会被包含在内，不管 `"exclude"` 如何设置。 如果没有特殊指定， `"exclude"` 默认情况下会排除 `node_modules` ， `bower_components` ， `jspm_packages` 和 `<outDir>` 目录。

任何被 `"files"` 或 `"include"` 指定的文件所引用的文件也会被包含进来。 `A.ts` 引用了 `B.ts` ，因此 `B.ts` 不能被排除，除非引用它的 `A.ts` 在 `"exclude"` 列表中。

需要注意编译器不会去引入那些可能做为输出的文件；比如，假设我们包含了 `index.ts` ，那么 `index.d.ts` 和 `index.js` 会被排除在外。 通常来讲，不推荐只有扩展名的不同来区分同目录下的文件。

`tsconfig.json` 文件可以是个空文件，那么所有默认的文件（如上面所述）都会以默认配置选项编译。

在命令行上指定的编译选项会覆盖在 `tsconfig.json` 文件里的相应选项。

`@types` ， `typeRoots` 和 `types`

默认所有可见的" `@types` "包会在编译过程中被包含进来。 `node_modules/@types` 文件夹下以及它们子文件夹下的所有包都是可见的； 也就是说， `./node_modules/@types/` ， `../node_modules/@types/` 和 `../../node_modules/@types/` 等等。

如果指定了 `typeRoots` ，只有 `typeRoots` 下面的包才会被包含进来。 比如：

```
{
   "compilerOptions": {
       "typeRoots" : ["./typings"]
   }
}
```

这个配置文件会包含所有 `./typings` 下面的包，而不包含 `./node_modules/@types` 里面的包。

如果指定了 `types` ，只有被列出来的包才会被包含进来。 比如：

```
{
   "compilerOptions": {
        "types" : ["node", "lodash", "express"]
   }
}
```

这个 `tsconfig.json` 文件将仅会包含 `./node_modules/@types/node` ， `./node_modules/@types/lodash` 和 `./node_modules/@types/express` 。/@types/。 `node_modules/@types/*` 里面的其它包不会被引入进来。

指定 `"types": []` 来禁用自动引入 `@types` 包。

注意，自动引入只在你使用了全局的声明（相反于模块）时是重要的。 如果你使用 `import "foo"` 语句，TypeScript仍然会查找 `node_modules` 和 `node_modules/@types` 文件夹来获取 `foo` 包。

使用 `extends` 继承配置

`tsconfig.json` 文件可以利用 `extends` 属性从另一个配置文件里继承配置。

`extends` 是 `tsconfig.json` 文件里的顶级属性（与 `compilerOptions` ， `files` ， `include` ，和 `exclude` 一样）。 `extends` 的值是一个字符串，包含指向另一个要继承文件的路径。

在原文件里的配置先被加载，然后被来至继承文件里的配置重写。 如果发现循环引用，则会报错。

来至所继承配置文件的 `files` ， `include` 和 `exclude` 覆盖源配置文件的属性。

配置文件里的相对路径在解析时相对于它所在的文件。

比如：

`configs/base.json` ：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

`tsconfig.json`：

```
{
  "extends": "./configs/base",
  "files": [
    "main.ts",
    "supplemental.ts"
  ]
}
```

`tsconfig.nostrictnull.json`：

```
{
  "extends": "./tsconfig",
  "compilerOptions": {
    "strictNullChecks": false
  }
}
```

`compileOnSave`

在最顶层设置 `compileOnSave` 标记，可以让IDE在保存文件的时候根据 `tsconfig.json` 重新生成文件。

```
{
    "compileOnSave": true,
    "compilerOptions": {
        "noImplicitAny" : true
    }
}
```

要想支持这个特性需要Visual Studio 2015， TypeScript1.8.4以上并且安装atom-typescript插件。

模式

到这里查看模式: http://json.schemastore.org/tsconfig.

> 本页面被移动到书写声明文件页

编译选项

选项	类型	默认值
`--allowJs`	`boolean`	`false`
`--allowSyntheticDefaultImports`	`boolean`	`module === "system"` 或 `--esModuleInterop`
`--allowUnreachableCode`	`boolean`	`false`
`--allowUnusedLabels`	`boolean`	`false`
`--alwaysStrict`	`boolean`	`false`
`--baseUrl`	`string`	
`--charset`	`string`	`"utf8"`
`--checkJs`	`boolean`	`false`
`--declaration` `-d`	`boolean`	`false`
`--declarationDir`	`string`	
`--diagnostics`	`boolean`	`false`
`--disableSizeLimit`	`boolean`	`false`
`--emitBOM`	`boolean`	`false`
`--emitDecoratorMetadata` [1]	`boolean`	`false`
`--experimentalDecorators` [1]	`boolean`	`false`
`--forceConsistentCasingInFileNames`	`boolean`	`false`
`--help` `-h`		
`--importHelpers`	`string`	
`--inlineSourceMap`	`boolean`	`false`
`--inlineSources`	`boolean`	`false`
`--init`		
`--isolatedModules`	`boolean`	`false`

`--jsx`	`string`	`"Preserve"`
`--jsxFactory`	`string`	`"React.createElem`
`--lib`	`string[]`	
`--listEmittedFiles`	`boolean`	`false`
`--listFiles`	`boolean`	`false`
`--locale`	`string`	*(platform specific)*

`--mapRoot`	`string`	
`--maxNodeModuleJsDepth`	`number`	`0`
`--module` `-m`	`string`	`target === "ES6"` `"ES6" : "commonjs"`

`--moduleResolution`	`string`	`module === "AMD"`
		`"System" or "ES6"`
		`"Classic" : "Node"`

`--newLine`	`string`	*(platform specific)*
`--noEmit`	`boolean`	`false`
`--noEmitHelpers`	`boolean`	`false`
`--noEmitOnError`	`boolean`	`false`
`--noErrorTruncation`	`boolean`	`false`
`--noFallthroughCasesInSwitch`	`boolean`	`false`
`--noImplicitAny`	`boolean`	`false`
`--noImplicitReturns`	`boolean`	`false`
`--noImplicitThis`	`boolean`	`false`
`--noImplicitUseStrict`	`boolean`	`false`
`--noLib`	`boolean`	`false`
`--noResolve`	`boolean`	`false`
`--noStrictGenericChecks`	`boolean`	`false`
`--noUnusedLocals`	`boolean`	`false`
`--noUnusedParameters`	`boolean`	`false`
~~`--out`~~	`string`	
`--outDir`	`string`	
`--outFile`	`string`	
`paths` [2]	`Object`	
`--preserveConstEnums`	`boolean`	`false`

`--preserveSymlinks`	`boolean`	`false`
`--preserveWatchOutput`	`boolean`	`false`
`--pretty` [1]	`boolean`	`false`
`--project` `-p`	`string`	
`--reactNamespace`	`string`	`"React"`
`--removeComments`	`boolean`	`false`
`--rootDir`	`string`	*(common root director computed from the lis input files)*
`rootDirs` [2]	`string[]`	
`--skipDefaultLibCheck`	`boolean`	`false`
`--skipLibCheck`	`boolean`	`false`
`--sourceMap`	`boolean`	`false`
`--sourceRoot`	`string`	
`--strict`	`boolean`	`false`
`--strictFunctionTypes`	`boolean`	`false`
`--strictPropertyInitialization`	`boolean`	`false`
`--strictNullChecks`	`boolean`	`false`
`--stripInternal` [1]	`boolean`	`false`
`--suppressExcessPropertyErrors` [1]	`boolean`	`false`
`--suppressImplicitAnyIndexErrors`	`boolean`	`false`
`--target` `-t`	`string`	`"ES3"`
`--traceResolution`	`boolean`	`false`

`--types`	`string[]`	
`--typeRoots`	`string[]`	
`--version` `-v`		
`--watch` `-w`		

- [1] 这些选项是试验性的。
- [2] 这些选项只能在 `tsconfig.json` 里使用，不能在命令行使用。

相关信息

- 在 `tsconfig.json` 文件里设置编译器选项。
- 在MSBuild工程里设置编译器选项。

概述

编译选项可以在使用MSBuild的项目里通过MSBuild属性指定。

例子

```
<PropertyGroup Condition="'$(Configuration)' == 'Debug'">
  <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
  <TypeScriptSourceMap>true</TypeScriptSourceMap>
</PropertyGroup>
<PropertyGroup Condition="'$(Configuration)' == 'Release'">
  <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
  <TypeScriptSourceMap>false</TypeScriptSourceMap>
</PropertyGroup>
<Import
    Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio
\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets"

    Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\Vi
sualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScri
pt.targets')" />
```

映射

编译选项	MSBuild属性名称
`--allowJs`	*MSBuild不支持此选项*
`--allowSyntheticDefaultImports`	TypeScriptAllowSyntheticDefaultImpo
`--allowUnreachableCode`	TypeScriptAllowUnreachableCode
`--allowUnusedLabels`	TypeScriptAllowUnusedLabels
`--alwaysStrict`	TypeScriptAlwaysStrict
`--baseUrl`	TypeScriptBaseUrl
`--charset`	TypeScriptCharset

`--declaration`	TypeScriptGeneratesDeclarations
`--declarationDir`	TypeScriptDeclarationDir
`--diagnostics`	*MSBuild*不支持此选项
`--disableSizeLimit`	*MSBuild*不支持此选项
`--emitBOM`	TypeScriptEmitBOM
`--emitDecoratorMetadata`	TypeScriptEmitDecoratorMetadata
`--experimentalAsyncFunctions`	TypeScriptExperimentalAsyncFunctio
`--experimentalDecorators`	TypeScriptExperimentalDecorators
`--forceConsistentCasingInFileNames`	TypeScriptForceConsistentCasingInF
`--help`	*MSBuild*不支持此选项
`--importHelpers`	TypeScriptImportHelpers
`--inlineSourceMap`	TypeScriptInlineSourceMap
`--inlineSources`	TypeScriptInlineSources
`--init`	*MSBuild*不支持此选项
`--isolatedModules`	TypeScriptIsolatedModules
`--jsx`	TypeScriptJSXEmit
`--jsxFactory`	TypeScriptJSXFactory
`--lib`	TypeScriptLib
`--listEmittedFiles`	*MSBuild*不支持此选项
`--listFiles`	*MSBuild*不支持此选项
`--locale`	*automatic*
`--mapRoot`	TypeScriptMapRoot
`--maxNodeModuleJsDepth`	*MSBuild*不支持此选项
`--module`	TypeScriptModuleKind
`--moduleResolution`	TypeScriptModuleResolution
`--newLine`	TypeScriptNewLine
`--noEmit`	*MSBuild*不支持此选项
`--noEmitHelpers`	TypeScriptNoEmitHelpers
`--noEmitOnError`	TypeScriptNoEmitOnError

`--noFallthroughCasesInSwitch`	TypeScriptNoFallthroughCasesInSwit
`--noImplicitAny`	TypeScriptNoImplicitAny
`--noImplicitReturns`	TypeScriptNoImplicitReturns
`--noImplicitThis`	TypeScriptNoImplicitThis
`--noImplicitUseStrict`	TypeScriptNoImplicitUseStrict
`--noStrictGenericChecks`	TypeScriptNoStrictGenericChecks
`--noUnusedLocals`	TypeScriptNoUnusedLocals
`--noUnusedParameters`	TypeScriptNoUnusedParameters
`--noLib`	TypeScriptNoLib
`--noResolve`	TypeScriptNoResolve
`--out`	TypeScriptOutFile
`--outDir`	TypeScriptOutDir
`--outFile`	TypeScriptOutFile
`--paths`	*MSBuild不支持此选项*
`--preserveConstEnums`	TypeScriptPreserveConstEnums
`--preserveSymlinks`	TypeScriptPreserveSymlinks
`--listEmittedFiles`	*MSBuild不支持此选项*
`--pretty`	*MSBuild不支持此选项*
`--reactNamespace`	TypeScriptReactNamespace
`--removeComments`	TypeScriptRemoveComments
`--rootDir`	TypeScriptRootDir
`--rootDirs`	*MSBuild不支持此选项*
`--skipLibCheck`	TypeScriptSkipLibCheck
`--skipDefaultLibCheck`	TypeScriptSkipDefaultLibCheck
`--sourceMap`	TypeScriptSourceMap
`--sourceRoot`	TypeScriptSourceRoot
`--strict`	TypeScriptStrict
`--strictFunctionTypes`	TypeScriptStrictFunctionTypes
`--strictNullChecks`	TypeScriptStrictNullChecks
`--stripInternal`	TypeScriptStripInternal

`--suppressExcessPropertyErrors`	TypeScriptSuppressExcessPropertyE
`--suppressImplicitAnyIndexErrors`	TypeScriptSuppressImplicitAnyIndexE
`--target`	TypeScriptTarget
`--traceResolution`	*MSBuild不支持此选项*
`--types`	*MSBuild不支持此选项*
`--typeRoots`	*MSBuild不支持此选项*
`--watch`	*MSBuild不支持此选项*
MSBuild only option	TypeScriptAdditionalFlags

我使用的Visual Studio版本里支持哪些选项?

查找 `C:\Program Files (x86)\MSBuild\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets` 文件。 可用的MSBuild XML标签与相应的 `tsc` 编译选项的映射都在那里。

ToolsVersion

工程文件里的 `<TypeScriptToolsVersion>1.7</TypeScriptToolsVersion>` 属性值表明了构建时使用的编译器的版本号（这个例子里是1.7） 这样就允许一个工程在不同的机器上使用相同版本的编译器进行构建。

如果没有指定 `TypeScriptToolsVersion` ，则会使用机器上安装的最新版本的编译器去构建。

如果用户使用的是更新版本的TypeScript，则会在首次加载工程的时候看到一个提示升级工程的对话框。

TypeScriptCompileBlocked

如果你使用其它的构建工具（比如，gulp， grunt等等）并且使用VS做为开发和调试工具，那么在工程里设置 `<TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>` 。 这样VS只会提供给你编辑的功能，而不会在你按F5的时候去构建。

Build tools

- Browserify
- Duo
- Grunt
- Gulp
- Jspm
- Webpack
- MSBuild
- NuGet

Browserify

安装

```
npm install tsify
```

使用命令行交互

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

使用API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
    .add('main.ts')
    .plugin('tsify', { noImplicitAny: true })
    .bundle()
    .pipe(process.stdout);
```

更多详细信息：smrq/tsify

Duo

安装

```
npm install duo-typescript
```

使用命令行交互

```
duo --use duo-typescript entry.ts
```

使用API

```
var Duo = require('duo');
var fs = require('fs')
var path = require('path')
var typescript = require('duo-typescript');

var out = path.join(__dirname, "output.js")

Duo(__dirname)
    .entry('entry.ts')
    .use(typescript())
    .run(function (err, results) {
        if (err) throw err;
        // Write compiled result to output file
        fs.writeFileSync(out, results.code);
    });
```

更多详细信息：frankwallis/duo-typescript

Grunt

安装

```
npm install grunt-ts
```

基本Gruntfile.js

```
module.exports = function(grunt) {
    grunt.initConfig({
        ts: {
            default : {
                src: ["**/*.ts", "!node_modules/**/*.ts"]
            }
        }
    });
    grunt.loadNpmTasks("grunt-ts");
    grunt.registerTask("default", ["ts"]);
};
```

更多详细信息：TypeStrong/grunt-ts

Gulp

安装

```
npm install gulp-typescript
```

基本gulpfile.js

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
    var tsResult = gulp.src("src/*.ts")
        .pipe(ts({
              noImplicitAny: true,
              out: "output.js"
        }));
    return tsResult.js.pipe(gulp.dest('built/local'));
});
```

更多详细信息：ivogabe/gulp-typescript

Jspm

安装

```
npm install -g jspm@beta
```

注意：目前*jspm*的*0.16beta*版本支持*TypeScript*

更多详细信息：TypeScriptSamples/jspm

Webpack

安装

```
npm install ts-loader --save-dev
```

基本webpack.config.js

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
        filename: "bundle.js"
    },
    resolve: {
        // Add '.ts' and '.tsx' as a resolvable extension.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx"
, ".js"]
    },
    module: {
        loaders: [
            // all files with a '.ts' or '.tsx' extension will b
e handled by 'ts-loader'
            { test: /\.tsx?$/, loader: "ts-loader" }
        ]
    }
};
```

查看更多关于ts-loader的详细信息

或者

- awesome-typescript-loader

MSBuild

更新工程文件，包含本地安装的 `Microsoft.TypeScript.Default.props` （在顶端）和 `Microsoft.TypeScript.targets` （在底部）文件：

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http:/
/schemas.microsoft.com/developer/msbuild/2003">
  <!-- Include default props at the top -->
  <Import
      Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio
\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.Default
.props"
      Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\Vi
sualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScri
pt.Default.props')" />

  <!-- TypeScript configurations go here -->
  <PropertyGroup Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Include default targets at the bottom -->
  <Import
      Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio
\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.targets"

      Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\Vi
sualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScri
pt.targets')" />
</Project>
```

关于配置MSBuild编译器选项的更多详细信息，请参考：在MSBuild里使用编译选项

NuGet

- 右键点击 -> Manage NuGet Packages
- 查找 `Microsoft.TypeScript.MSBuild`
- 点击 `Install`
- 安装完成后，Rebuild。

更多详细信息请参考Package Manager Dialog和using nightly builds with NuGet

在太平洋标准时间每天午夜会自动构建TypeScript的 `master` 分支代码并发布到NPM和NuGet上。 下面将介绍如何获得并在工具里使用它们。

使用 npm

```
npm install -g typescript@next
```

使用 NuGet 和 MSBuild

> 注意：你需要配置工程来使用NuGet包。 详细信息参考配置MSBuild工程来使用NuGet。

www.myget.org。

有两个包：

- `Microsoft.TypeScript.Compiler`：仅包含工具（`tsc.exe`，`lib.d.ts`，等。）。
- `Microsoft.TypeScript.MSBuild`：和上面一样的工具，还有MSBuild的任务和目标（`Microsoft.TypeScript.targets`，`Microsoft.TypeScript.Default.props`，等。）

更新IDE来使用每日构建

你还可以配置IDE来使用每日构建。 首先你要通过npm安装包。 你可以进行全局安装或者安装到本地的 `node_modules` 目录下。

下面的步骤里我们假设你已经安装好了 `typescript@next`。

Visual Studio Code

更新 `.vscode/settings.json` 如下：

```
"typescript.tsdk": "<path to your folder>/node_modules/typescrip
t/lib"
```

详细信息参见VSCode文档。

Sublime Text

更新 `Settings - User` 如下：

```
"typescript_tsdk": "<path to your folder>/node_modules/typescrip
t/lib"
```

详细信息参见如何在Sublime Text里安装TypeScript插件。

Visual Studio 2013 and 2015

> 注意：大多数的改变不需要你安装新版本的VS TypeScript插件。

当前的每日构建不包含完整的插件安装包，但是我们正在试着提供每日构建的安装包。

1. 下载VSDevMode.ps1脚本。

   > 参考wiki文档：使用自定义语言服务文件。

2. 在PowerShell命令行窗口里执行：

   VS 2015：

   ```
   VSDevMode.ps1 14 -tsScript <path to your folder>/node_module
   s/typescript/lib
   ```

   VS 2013：

   ```
   VSDevMode.ps1 12 -tsScript <path to your folder>/node_module
   s/typescript/lib
   ```

IntelliJ IDEA (Mac)

前往 `Preferences` > `Languages & Frameworks` > `TypeScript` :

> TypeScript Version: 如果通过NPM安装: `/usr/local/lib/node_modules/typescript/lib`

Table of Contents

介绍

在JavaScript里（还有TypeScript）， `this` 关键字的行为与其它语言相比大为不同。这可能会很令人吃惊，特别是对于那些使用其它语言的用户，他们凭借其直觉来想象 `this` 关键字的行为。

这篇文章会教你怎么识别及调试TypeScript里的 `this` 问题，并且提供了一些解决方案和各自的利弊。

典型症状和危险系数

丢失 `this` 上下文的典型症状包括：

- 类的某字段（ `this.foo` ）为 `undefined` ，但其它值没有问题
- `this` 的值指向全局的 `window` 对象而不是类实例对象（在非严格模式下）
- `this` 的值为 `undefined` 而不是类实例对象（严格模式下）
- 调用类方法（ `this.doBa()` ）失败，错误信息如“TypeError: undefined is not a function”，“Object doesn't support property or method 'doBar'”或“this.doBar is not a function”

程序中应该出现了以下代码：

- 事件监听，比如 `window.addEventListener('click', myClass.doThing);`
- Promise解决，比如 `myPromise.then(myClass.theNextThing);`
- 第三方库回调，比如 `$(document).ready(myClass.start);`
- 函数回调，比如 `someArray.map(myClass.convert)`
- ViewModel类型的库里的类，比如 `<div data-bind="click: myClass.doSomething">`
- 可选包里的函数，比如 `$.ajax(url, { success: myClass.handleData })`

JavaScript里的 `this` 究竟是什么？

已经有大量的文章讲述了JavaScript里 `this` 关键字的危险性。查看这里，这里，或这里。

当JavaScript里的一个函数被调用时，你可以按照下面的顺序来推断出 `this` 指向的是什么（这些规则是按优先级顺序排列的）：

- 如果这个函数是 `function#bind` 调用的结果，那么 `this` 指向的是传入 `bind` 的参数
- 如果函数是以 `foo.func()` 形式调用的，那么 `this` 值为 `foo`
- 如果是在严格模式下， `this` 将为 `undefined`
- 否则， `this` 将是全局对象（浏览器环境里为 `window` ）

这些规则会产生与直觉相反的效果。比如：

```
class Foo {
  x = 3;
  print() {
    console.log('x is ' + this.x);
  }
}

var f = new Foo();
f.print(); // Prints 'x is 3' as expected

// Use the class method in an object literal
var z = { x: 10, p: f.print };
z.p(); // Prints 'x is 10'

var p = z.p;
p(); // Prints 'x is undefined'
```

`this` 的危险信号

你要注意的最大的危险信号是在要使用类的方法时没有立即调用它。任何时候你看到类方法被引用了却没有使用相同的表达式来调用时， `this` 可能已经不对了。

例子：

```
var x = new MyObject();
x.printThing(); // SAFE, method is invoked where it is referenced


var y = x.printThing; // DANGER, invoking 'y()' may not have cor
rect 'this'

window.addEventListener('click', x.printThing, 10); // DANGER, m
ethod is not invoked where it is referenced

window.addEventListener('click', () => x.printThing(), 10); // S
AFE, method is invoked in the same expression
```

修复

可以通过一些方法来保持 `this` 的上下文。

使用实例函数

代替TypeScript里默认的原型方法，你可以使用一个实例箭头函数来定义类成员：

```
class MyClass {
    private status = "blah";

    public run = () => { // <-- note syntax here
        alert(this.status);
    }
}
var x = new MyClass();
$(document).ready(x.run); // SAFE, 'run' will always have correc
t 'this'
```

- 好与坏：这会为每个类实例的每个方法创建额外的闭包。如果这个方法通常是正常调用的，那么这么做有点过了。然而，它经常会在回调函数里调用，让类实例捕获到 `this` 上下文会比在每次调用时都创建一个闭包来得更有效率一些。

- 好：其它外部使用者不可能忘记处理 `this` 上下文
- 好：在TypeScript里是类型安全的
- 好：如果函数带参数不需要额外的工作
- 坏：派生类不能通过使用 `super` 调用基类方法
- 坏：在类与用户之前产生了额外的非类型安全的约束：明确了哪些方法提前绑定了以及哪些没有

本地的胖箭头

在TypeScrip里（这里为了讲解添加了一些参数）：

```
var x = new SomeClass();
someCallback((n, m) => x.doSomething(n, m));
```

- 好与坏：内存/效能上的利弊与实例函数相比正相反
- 好：在TypeScript，100%的类型安全
- 好：在ECMAScript 3里同样生效
- 好：你只需要输入一次实例名
- 坏：你要输出2次参数名
- 坏：对于可变参数不起作用（'rest'）

Function.bind

```
var x = new SomeClass();
// SAFE: Functions created from function.bind are always preserv
e 'this'
window.setTimeout(x.someMethod.bind(x), 100);
```

- 好与坏：内存/效能上的利弊与实例函数相比正相反
- 好：如果函数带参数不需要额外的工作
- 坏：目前在TypeScript里，不是类型安全的
- 坏：只在ECMAScript 5里生效
- 坏：你要输入2次实例名

这个编码规范是给TypeScript开发团队在开发TypeScript时使用的。对于使用TypeScript的普通用户来说不一定适用，但是可以做为一个参考。

命名

1. 使用PascalCase为类型命名。
2. 不要使用`I`做为接口名前缀。
3. 使用PascalCase为枚举值命名。
4. 使用camelCase为函数命名。
5. 使用camelCase为属性或本地变量命名。
6. 不要为私有属性名添加`_`前缀。
7. 尽可能使用完整的单词拼写命名。

组件

1. 1个文件对应一个逻辑组件（比如：解析器，检查器）。
2. 不要添加新的文件。:)
3. `.generated.*`后缀的文件是自动生成的，不要手动改它。

类型

1. 不要导出类型/函数，除非你要在不同的组件中共享它。
2. 不要在全局命名空间内定义类型/值。
3. 共享的类型应该在`types.ts`里定义。
4. 在一个文件里，类型定义应该出现在顶部。

`null` 和 `undefined`：

1. 使用 **undefined**，不要使用 null。

一般假设

1. 假设像Nodes，Symbols等这样的对象在定义它的组件外部是不可改变的。不要去改变它们。

2. 假设数组是不能改变的。

类

1. 为了保持一致，在核心编译链中不要使用类，使用函数闭包代替。

标记

1. 一个类型中有超过2个布尔属性时，把它变成一个标记。

注释

为函数，接口，枚举类型和类使用JSDoc风格的注释。

字符串

1. 使用双引号 `""`
2. 所有要展示给用户看的信息字符串都要做好本地化工作（在diagnosticMessages.json中创建新的实体）。

错误提示信息

1. 在句子结尾使用 `.` 。
2. 对不确定的实体使用不定冠词。
3. 确切的实体应该使用名字（变量名，类型名等）
4. 当创建一条新的规则时，主题应该使用单数形式（比如：An external module cannot...而不是External modules cannot）。
5. 使用现在时态。

错误提示信息代码

提示信息被划分类成了一般的区间。如果要新加一个提示信息，在上条代码上加1做为新的代码。

- 1000 语法信息
- 2000 语言信息
- 4000 声明生成信息
- 5000 编译器选项信息
- 6000 命令行编译器信息
- 7000 noImplicitAny信息

普通方法

由于种种原因，我们避免使用一些方法，而使用我们自己定义的。

1. 不使用ECMAScript 5函数；而是使用core.ts这里的。
2. 不要使用 `for..in` 语句；而是使用 `ts.forEach` ，`ts.forEachKey` 和 `ts.forEachValue` 。注意它们之间的区别。
3. 如果可能的话，尝试使用 `ts.forEach` ，`ts.map` 和 `ts.filter` 代替循环。

风格

1. 使用arrow函数代替匿名函数表达式。
2. 只要需要的时候才把arrow函数的参数括起来。
   比如，`(x) => x + x` 是错误的，下面是正确的做法：
   i. `x => x + x`
   ii. `(x,y) => x + y`
   iii. `<T>(x: T, y: T) => x === y`
3. 总是使用 `{}` 把循环体和条件语句括起来。
4. 开始的 `{` 总是在同一行。
5. 小括号里开始不要有空白.
   逗号，冒号，分号后要有一个空格。比如：
   i. `for (var i = 0, n = str.length; i < 10; i++) { }`
   ii. `if (x < 10) { }`
   iii. `function f(x: number, y: string): void { }`

6. 每个变量声明语句只声明一个变量（比如 使用 `var x = 1; var y = 2;` 而不是 `var x = 1, y = 2;`）。
7. `else` 要在结束的 `}` 后另起一行。

介绍

下面列出了一些在使用TypeScript语言和编译器过程中常见的容易让人感到困惑的错误信息。

令人困惑的常见错误

"tsc.exe" exited with error code 1

修复：

- 检查文件编码，确保为UTF-8 - https://typescript.codeplex.com/workitem/1587

external module XYZ cannot be resolved

修复：

- 检查模块路径是否大小写敏感 - https://typescript.codeplex.com/workitem/2134

快捷列表

- Atom
- Eclipse
- Emacs
- NetBeans
- Sublime Text
- TypeScript Builder
- Vim
- Visual Studio
- Visual Studio Code
- WebStorm

Atom

Atom-TypeScript，由TypeStrong开发的针对Atom的TypeScript语言服务。

Eclipse

Eclipse TypeScript 插件，由Palantir开发的Eclipse插件。

Emacs

tide - TypeScript Interactive Development Environment for Emacs

NetBeans

- nbts - NetBeans TypeScript editor plugin
- Geertjan's TypeScript NetBeans Plugin

Sublime Text

Sublime的TypeScript插件，可以通过Package Control来安装，支持Sublime Text 2和Sublime Text 3.

TypeScript Builder

TypeScript Builder，TypeScript专用IDE.

Vim

语法高亮

- leafgarland/typescript-vim提供了语法文件用来高亮显示 `.ts` 和 `.d.ts` 。
- HerringtonDarkholme/yats.vim提供了更多语法高亮和DOM关键字。

语言服务工具

有两个主要的TypeScript插件：

- Quramy/tsuquyomi
- clausreinke/typescript-tools.vim

如果你想要输出时自动补全功能，你可以安装YouCompleteMe并添加以下代码到 `.vimrc` 里，以指定哪些符号能用来触发补全功能。YouCompleteMe会调用它们各自TypeScript插件来进行语义查询。

```
if !exists("g:ycm_semantic_triggers")
  let g:ycm_semantic_triggers = {}
endif
let g:ycm_semantic_triggers['typescript'] = ['.']
```

Visual Studio 2013/2015

Visual Studio里安装Microsoft Web Tools时就带了TypeScript。

TypeScript for Visual Studio 2015 在这里

TypeScript for Visual Studio 2013 在这里

Visual Studio Code

Visual Studio Code，是一个轻量级的跨平台编辑器，内置了对TypeScript的支持。

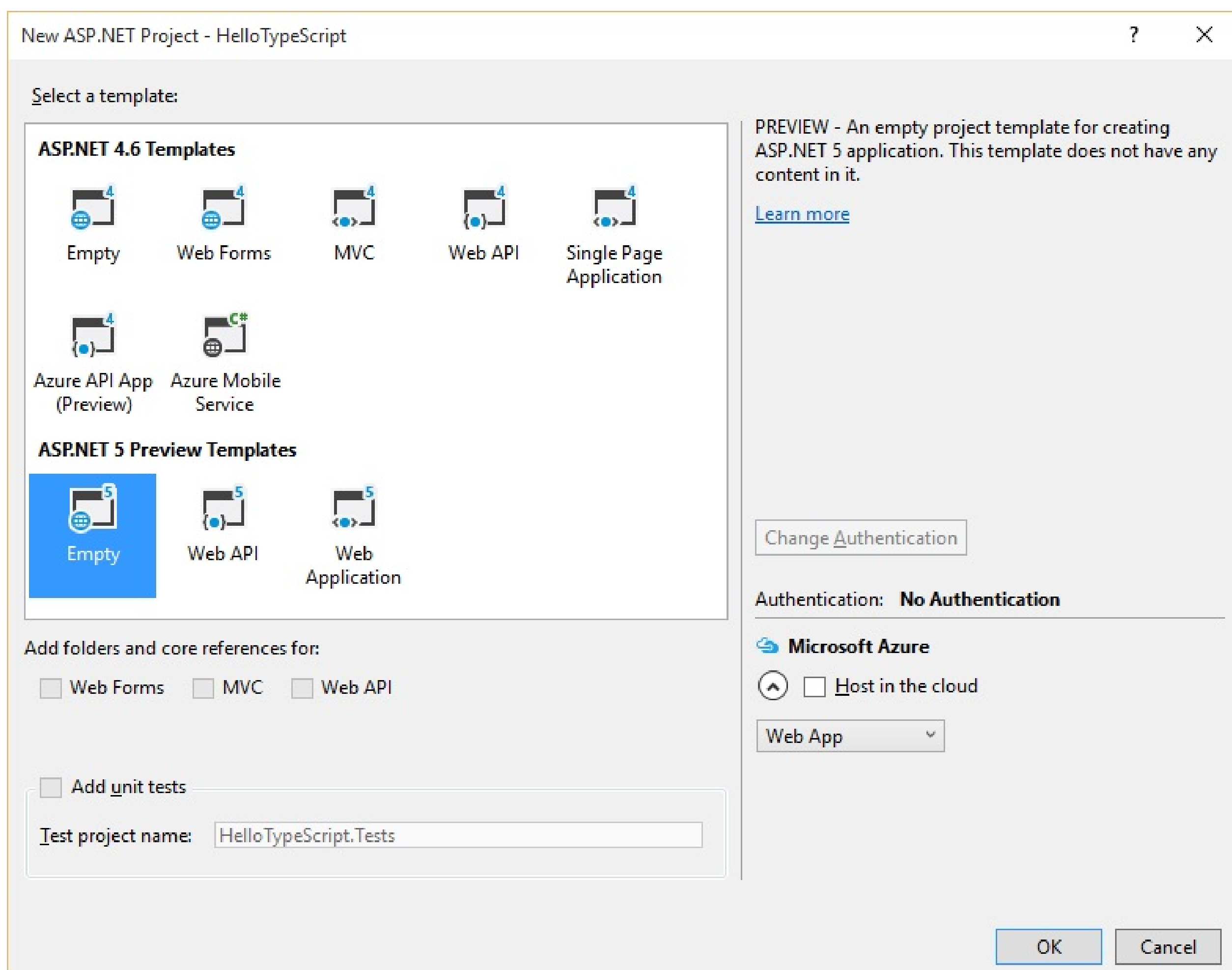
Webstorm

WebStorm，同其它JetBrains IDEs一样，直接包含了对TypeScript的支持。

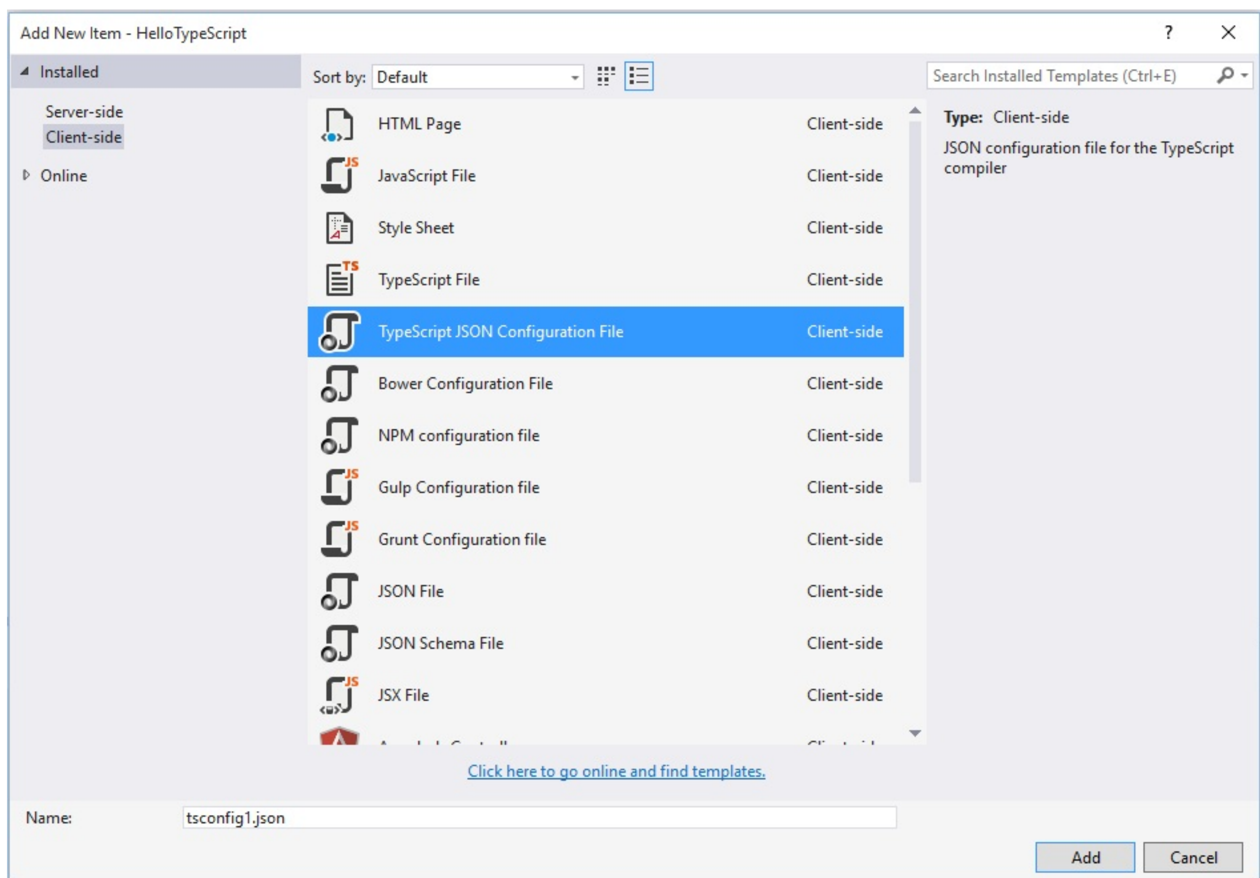
与ASP.NET v5一起使用TypeScript需要你用特定的方式来设置你的工程。 更多关于ASP.NET v5的详细信息请查看ASP.NET v5 文档 在Visual Studio的工程里支持当前的tsconfig.json还在开发之中，可以在这里查看进度#3983。

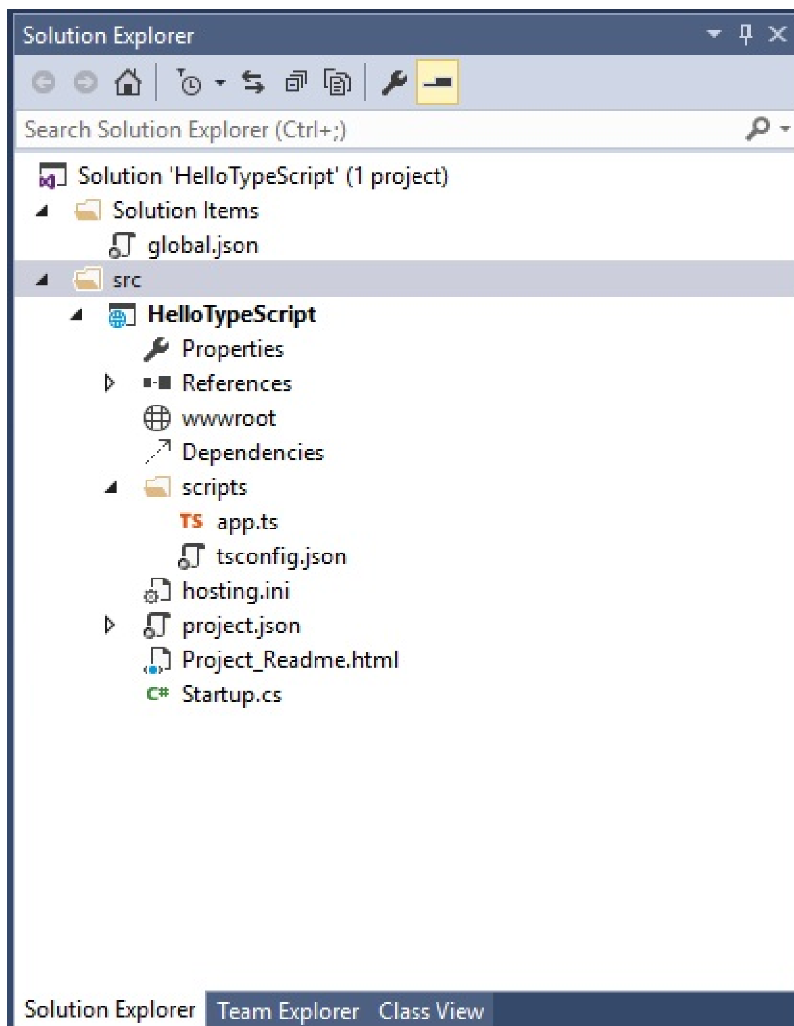
工程设置

我们就以在Visual Studio 2015里创建一个空的ASP.NET v5工程开始，如果你对ASP.NET v5还不熟悉，可以查看这个教程。



然后在工程根目录下添加一个 `scripts` 目录。 这就是我们将要添加TypeScript文件和 `tsconfig.json` 文件来设置编译选项的地方。 请注意目录名和路径都必须这样才能正常工作。 添加 `tsconfig.json` 文件，右键点击 `scripts` 目录，选择 `Add` ， `New Item` 。 在 `Client-side` 下，你能够找到它，如下所示。


Add New Item - HelloTypeScript
Installed
Server-side
Client-side
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
HTML Page Client-side
JavaScript File Client-side
Style Sheet Client-side
TypeScript File Client-side
TypeScript JSON Configuration File Client-side
Bower Configuration File Client-side
NPM configuration file Client-side
Gulp Configuration file Client-side
Grunt Configuration file Client-side
JSON File Client-side
JSON Schema File Client-side
JSX File Client-side
Type: Client-side
JSON configuration file for the TypeScript compiler
Click here to go online and find templates.
Name: tsconfig1.json
Add
Cancel


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloTypeScript' (1 project)
Solution Items
global.json
src
HelloTypeScript
Properties
References
wwwroot
Dependencies
scripts
app.ts
tsconfig.json
hosting.ini
project.json
Project_Readme.html
Startup.cs
Solution Explorer
Team Explorer
Class View

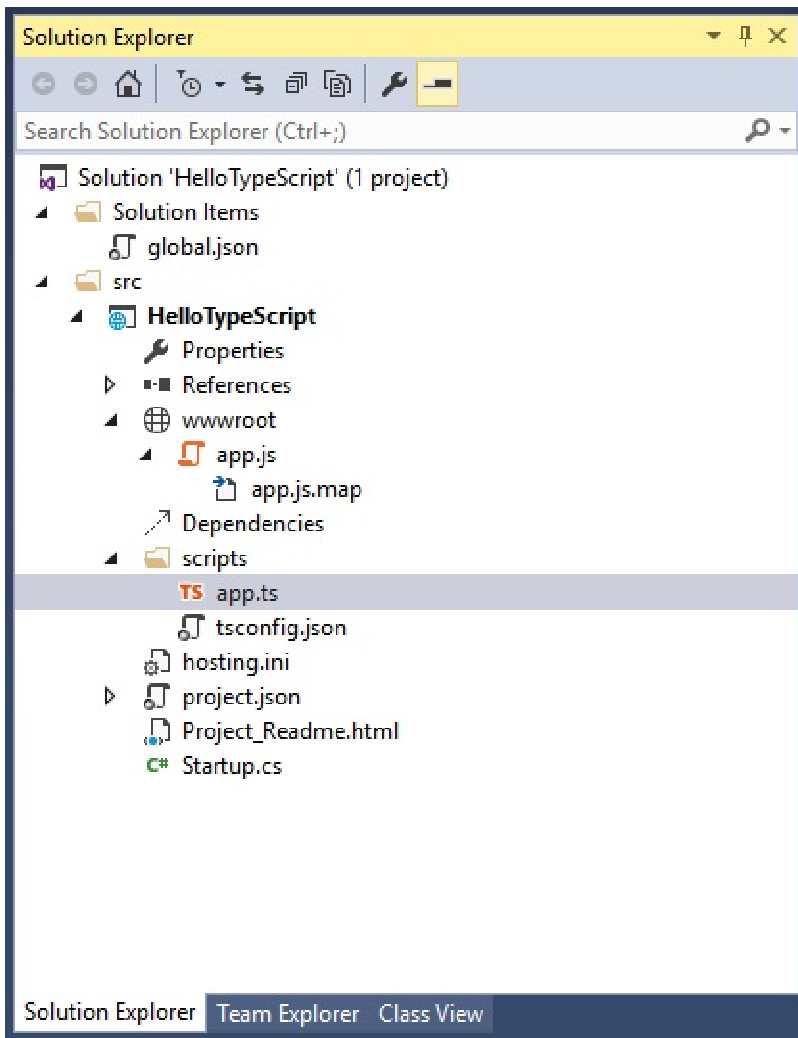
最后我们还要将下面的选项添加到 `tsconfig.json` 文件的 `"compilerOptions"` 节点里，让编译器输出重定向到 `wwwroot` 文件夹：

```
"outDir": "../wwwroot/"
```

下面是配置好 `tsconfig.json` 后可能的样子

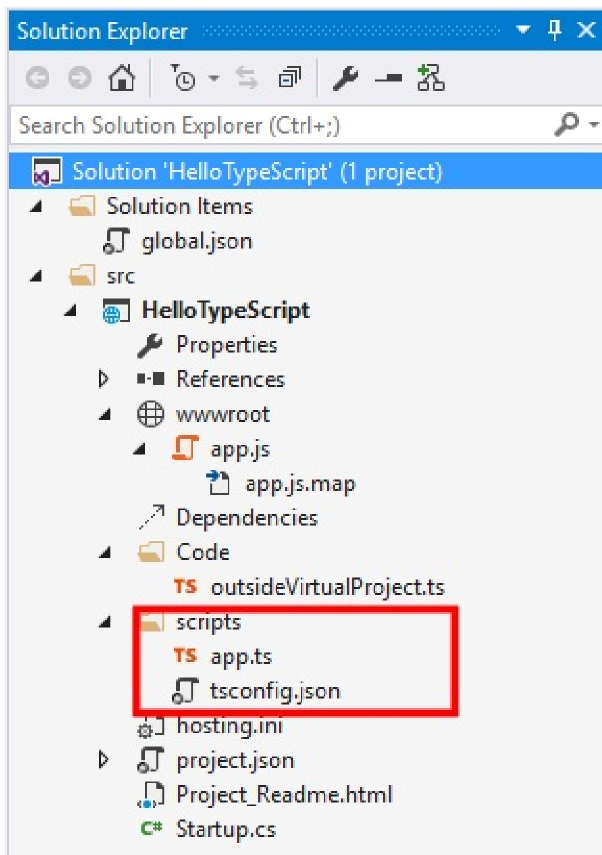
```
{
    "compilerOptions": {
        "noImplicitAny": false,
        "noEmitOnError": true,
        "removeComments": false,
        "sourceMap": true,
        "target": "es5",
        "outDir": "../wwwroot"
    }
}
```

现在如果我们构建这个工程，你就会注意到 `app.js` 和 `app.js.map` 文件被创建在 `wwwroot` 目录里。



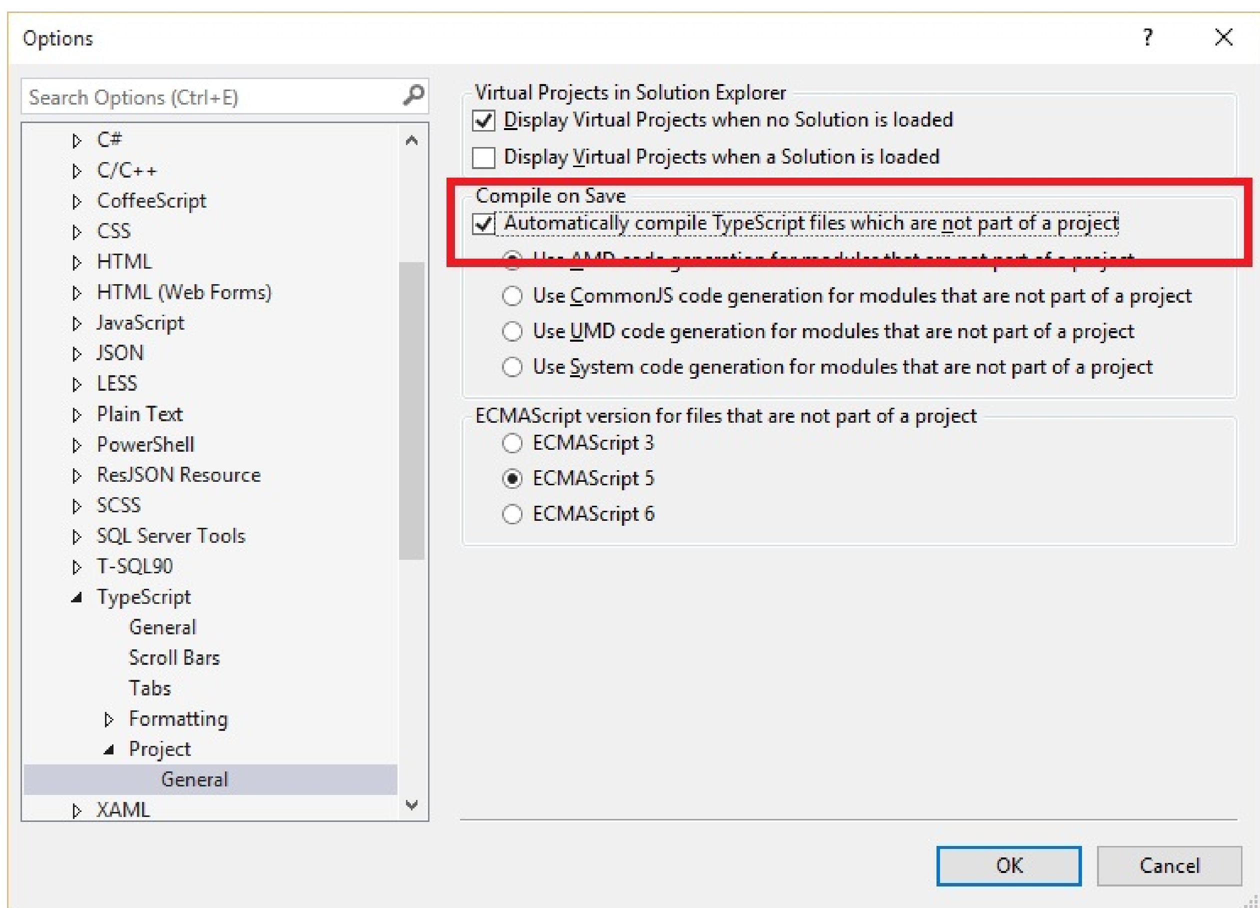
工程与虚拟工程

当添加了一个 `tsconfig.json` 文件，你要明白很重要的一点是我们创建了一个虚拟TypeScript工程，在包含 `tsconfig.json` 文件的目录下。被当作这个虚拟工程一部分的TypeScript文件是不会在保存的时候编译的。在包含 `tsconfig.json` 文件的目录外层里存在的TypeScript文件不会被当作虚拟工程的一部分。下图中，可以见到这个虚拟工程，在红色矩形里。

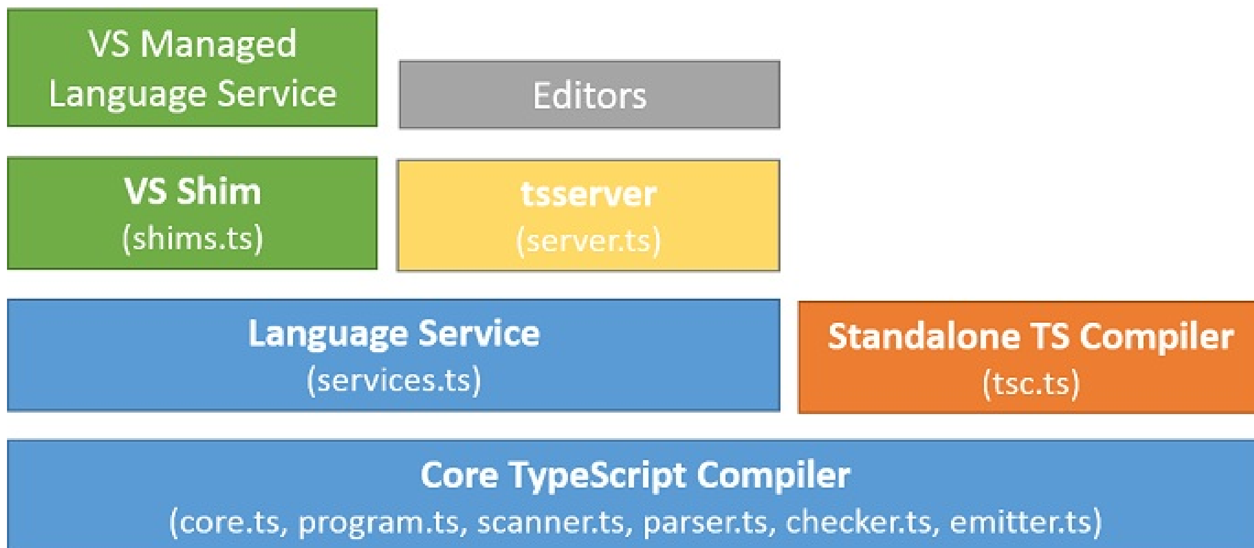


保存时编译

想要启用ASP.NET v5项目的保存时编译功能，你必须为不是虚拟TypeScript工程一部分的TypeScript文件启用保存时编译功能。如果工程里存在 `tsconfig.json` 文件，那么模块类型选项的设置会被忽略。


Options
Search Options (Ctrl+E)
C#
C/C++
CoffeeScript
CSS
HTML
HTML (Web Forms)
JavaScript
JSON
LESS
Plain Text
PowerShell
ResJSON Resource
SCSS
SQL Server Tools
T-SQL90
TypeScript
General
Scroll Bars
Tabs
Formatting
Project
General
XAML
Virtual Projects in Solution Explorer
Display Virtual Projects when no Solution is loaded
Display Virtual Projects when a Solution is loaded
Compile on Save
Automatically compile TypeScript files which are not part of a project
Use CommonJS code generation for modules that are not part of a project
Use UMD code generation for modules that are not part of a project
Use System code generation for modules that are not part of a project
ECMAScript version for files that are not part of a project
ECMAScript 3
ECMAScript 5
ECMAScript 6
OK
Cancel

层次概述



- 核心**TypeScript**编译器
    - 语法分析器（**Parser**）：以一系列原文件开始, 根据语言的语法, 生成抽象语法树（AST）
    - 联合器（**Binder**）：使用一个 `Symbol` 将针对相同结构的声明联合在一起（例如：同一个接口或模块的不同声明，或拥有相同名字的函数和模块）。这能帮助类型系统推导出这些具名的声明。
    - 类型解析器与检查器（**Type resolver / Checker**）：解析每种类型的构造，检查读写语义并生成适当的诊断信息。
    - 生成器（**Emitter**）：从一系列输入文件（.ts和.d.ts）生成输出，它们可以是以下形式之一：JavaScript（.js），声明（.d.ts），或者是source maps（.js.map）。
    - 预处理器（**Pre-processor**）：“编译上下文”指的是某个“程序”里涉及到的所有文件。上下文的创建是通过检查所有从命令行上传入编译器的文件，按顺序，然后再加入这些文件直接引用的其它文件或通过 `import` 语句和 `/// <reference path=... />` 标签间接引用的其它文件。

沿着引用图走下来你会发现它是一个有序的源文件列表，它们组成了整个程序。 当解析导出（import）的时候，会优先选择“.ts”文件而不是“.d.ts”文件，以确保处理的是最新的文件。 编译器会进行与Nodejs相似的流程来解析导入，沿着目录链查找与

将要导入相匹配的带.ts或.d.ts扩展名的文件。 导入失败不会报error，因为可能已经声明了外部模块。

- 独立编译器（**tsc**）： 批处理编译命令行界面。主要处理针对不同支持的引擎读写文件（比如：Node.js）。
- 语言服务：“语言服务”在核心编译器管道上暴露了额外的一层，非常适合类编辑器的应用。

语言服务支持一系列典型的编辑器操作比如语句自动补全，函数签名提示，代码格式化和突出高亮，着色等。 基本的重构功能比如重命名，调试接口辅助功能比如验证断点，还有TypeScript特有的功能比如支持增量编译（在命令行上使用 `--watch` ）。 语言服务是被设计用来有效的处理在一个长期存在的编译上下文中文件随着时间改变的情况；在这样的情况下，语言服务提供了与其它编译器接口不同的角度来处理程序和源文件。

> 请参考 [[Using the Language Service API]] 以了解更多详细内容。

数据结构

- **Node:** 抽象语法树（AST）的基本组成块。通常 `Node` 表示语言语法里的非终结符；一些终结符保存在语法树里比如标识符和字面量。
- **SourceFile:** 给定源文件的AST。 `SourceFile` 本身是一个 `Node` ；它提供了额外的接口用来访问文件的源码，文件里的引用，文件里的标识符列表和文件里的某个位置与它对应的行号与列号的映射。
- **Program:** `SourceFile` 的集合和一系列编译选项代表一个编译单元。 `Program` 是类型系统和生成代码的主入口。
- **Symbol:** 具名的声明。 `Symbols` 是做为联合的结果而创建。 `Symbols` 连接了树里的声明节点和其它对同一个实体的声明。 `Symbols` 是语义系统的基本构建块。
- **Type:** `Type` 是语义系统的其它部分。 `Type` 可能被命名（比如，类和接口），或匿名（比如，对象类型）。
- **Signature:** 一共有三种 `Signature` 类型：调用签名（call），构造签名（construct）和索引签名（index）。

编译过程概述

整个过程从预处理开始。 预处理器会算出哪些文件参与编译，它会去查找如下引用（`/// <reference path=... />` 标签和 `import` 语句）。

语法分析器（Parser）生成抽象语法树（AST） `Node` . 这些仅为用户输出的抽象表现，以树的形式。 一个 `SourceFile` 对象表示一个给定文件的AST并且带有一些额外的信息如文件名及源文件内容。

然后，联合器（Binder）处理AST节点，结合并生成 `Symbols` 。 一个 `Symbol` 会对应到一个命名实体。 这里有个一微妙的差别，几个声明节点可能会是名字相同的实体。 也就是说，有时候不同的 `Node` 具有相同的 `Symbol` ，并且每个 `Symbol` 保持跟踪它的声明节点。 比如，一个名字相同的 `class` 和 `namespace` 可以合并，并且拥有相同的 `Symbol` 。 联合器也会处理作用域，以确保每个 `Symbol` 都在正确的封闭作用域里创建。

生成 `SourceFile` （还带有它的 `Symbols` 们）是通过调用 `createSourceFile` API。

到目前为止， `Symbol` 代表的命名实体可以在单个文件里看到，但是有些声明可以从多文件合并，因此下一步就是构建一个全局的包含所有文件的视图，也就是创建一个 `Program` 。

一个 `Program` 是 `SourceFile` 的集合并带有一系列 `CompilerOptions` 。 通过调用 `createProgram` API来创建 `Program` 。

通过一个 `Program` 实例创建 `TypeChecker` 。 `TypeChecker` 是TypeScript类型系统的核心。 它负责计算出不同文件里的 `Symbols` 之间的关系，将 `Type` 赋值给 `Symbol` ，并生成任何语义 `Diagnostic` （比如：error）。

`TypeChecker` 首先要做的是合并不同的 `SourceFile` 里的 `Symbol` 到一个单独的视图，创建单一的 `Symbol` 表，合并所有普通的 `Symbol` （比如：不同文件里的 `namespace` ）。

在原始状态初始化完成后， `TypeChecker` 就可以解决关于这个程序的任何问题了。 这些“问题”可以是：

- 这个 `Node` 的 `Symbol` 是什么？
- 这个 `Symbol` 的 `Type` 是什么？
- 在AST的某个部分里有哪些 `Symbol` 是可见的？

- 某个函数声明的 `Signature` 都有哪些？
- 针对某个文件应该报哪些错误？

`TypeChecker` 计算所有东西都是“懒惰的”；为了回答一个问题它仅“解决”必要的信息。 `TypeChecker` 仅会检测和这个问题有关的 `Node` ， `Symbol` 或 `Type` ，不会检测额外的实体。

对于一个 `Program` 同样会生成一个 `Emitter` 。 `Emitter` 负责生成给定 `SourceFile` 的输出；它包括： `.js` ， `.jsx` ， `.d.ts` 和 `.js.map` 。

术语

完整开始/令牌开始（**Full Start/Token Start**）

令牌本身就具有我们称为一个“完整开始”和一个“令牌开始”。“令牌开始”是指更自然的版本，它表示在文件中令牌开始的位置。“完整开始”是指从上一个有意义的令牌之后扫描器开始扫描的起始位置。当关心琐事时，我们往往更关心完整开始。

函数	描述
`ts.Node.getStart`	取得某节点的第一个令牌起始位置。
`ts.Node.getFullStart`	取得某节点拥有的第一个令牌的完整开始。

琐碎内容（**Trivia**）

语法的琐碎内容代表源码里那些对理解代码无关紧要的内容，比如空白，注释甚至一些冲突的标记。

因为琐碎内容不是语言正常语法的一部分（不包括ECMAScript API规范）并且可能在任意2个令牌中的任意位置出现，它们不会包含在语法树里。但是，因为它们对于像重构和维护高保真源码很重要，所以需要的时候还是能够通过我们的APIs访问。

因为 `EndOfFileToken` 后面可以没有任何内容（令牌和琐碎内容），所有琐碎内容自然地在非琐碎内容之前，而且存在于那个令牌的“完整开始”和“令牌开始”之间。

虽然这个一个方便的标记法来说明一个注释“属于”一个 `Node` 。比如，在下面的例子里，可以明显看出 `genie` 函数拥有两个注释：

```
var x = 10; // This is x.

/**
 * Postcondition: Grants all three wishes.
 */
function genie([wish1, wish2, wish3]: [Wish, Wish, Wish]) {
    while (true) {
    }
} // End function
```

这是尽管事实上，函数声明的完整开始是在 `var x = 10;` 后。

我们依据Roslyn's notion of trivia ownership处理注释所有权。通常来讲，一个令牌拥有同一行上的所有的琐碎内容直到下一个令牌开始。任何出现在这行之后的注释都属于下一个令牌。源文件的第一个令牌拥有所有的初始琐碎内容，并且最后面的一系列琐碎内容会添加到 `end-of-file` 令牌上。

对于大多数普通用户，注释是“有趣的”琐碎内容。属于一个节点的注释内容可以通过下面的函数来获取：

函数	描述
`ts.getLeadingCommentRanges`	提供源文件和一个指定位置，返回指定位置后的第一个换行与令牌之间的注释的范围（与 `ts.Node.getFullStart` 配合会更有用）。
`ts.getTrailingCommentRanges`	提供源文件和一个指定位置，返回到指定位置后第一个换行为止的注释的范围（与 `ts.Node.getEnd` 配合会更有用）。

做为例子，假设有下面一部分源代码：

```
debugger;/*hello*/
    //bye
  /*hi*/    function
```

`function` 关键字的完整开始是从 `/*hello*/` 注释，但是 `getLeadingCommentRanges` 仅会返回后面2个注释：

```
d e b u g g e r ; / * h e l l o * / _ _ _ _ _ _ [CR] [NL] _ _ _ _
/ / b y e [CR] [NL] _ _ / * h i * / _ _ _ _ f u n c t i o n
                  ↑                                    ↑
↑                       ↑                   ↑
                 完整开始                              查找
第一个注释                第二个注释    令牌开始
                                                      开始注释
```

适当地，在 `debugger` 语句后调用 `getTrailingCommentRanges` 可以提取出 `/*hello*/` 注释。

如果你关心令牌流的更多信息， `createScanner` 也有一个 `skipTrivia` 标记，你可以设置成 `false` ，然后使用 `setText` / `setTextPos` 来扫描文件里的不同位置。

2.1

- 调查 Function bind 操作符
- 支持工程引用
- `readonly` 修饰符
- 调查 具名类型支持
- Language Service API里支持代码重构功能
- 扁平化声明

2.0

- 切换到基于转换的生成器
- 支持ES5/ES3 `async` / `await`
- 支持ES7对象属性展开及剩余属性
- 规定函数的 `this` 类型
- 属性访问上的类型保护
- 切换类型保护
- 支持常量和Symbol上计算属性的类型检查
- 可变类型
- 外部装饰器
- 弃用的装饰器
- 条件装饰器
- 函数表达式及箭头函数的装饰器
- 支持节点注册勾子
- 在tsconfig.json里支持Glob
- 在语言服务API里支持快速修复
- 在tsserver/语言服务API里集成tsd
- 从js文件的JSDoc里撮类型信息
- 增强lib.d.ts模块化
- 支持外部辅助代码库
- 调查语言服务的可扩展性

1.8

- 在TypeScript编译时使用 `--allowjs` 允许JavaScript
- 在循环里允许捕获的 `let` / `const`
- 标记死代码
- 使用 `--outFile` 连接模块输出
- tsconfig.json里支持注释
- 使用 `--pretty` 为终端里的错误信息添加样式
- 支持 `--outFile` 给命名的管道套接字和特殊设备
- 支持使用名字字面量的计算属性
- 字符串字面量类型
- JSX无状态的功能性组件
- 优化联合/交类型接口
- 支持F-Bounded多态性
- 支持全路径 `-project` / `-p` 参数
- 在SystemJS使用 `--allowSyntheticDefaultImports` 支持 `default` 导入操作
- 识别JavaScript里原型的赋值
- 在模块里使用路径映射
- 在其它模块里增加global/module作用域
- 在Visual Studio使用tsconfig.json做为高优先级的配置
- 基于 `this` 类型保护
- 支持自定义JSX工厂通过 `--reactNamespace`
- 增强for-in语句检查
- JSX代码在VS 2015里高亮
- 发布TypeScript NuGet 包

1.7

- ES7幂运算符
- 多态的 `this` 类型
- 支持 `--module` 的 `--target es6`
- 支持目标为ES3时使用装饰器
- 为ES6支持 `async` / `await` (Node v4)
- 增强的字面量初始化器解构检查

1.6

- ES6 Generators
- Local types
- 泛型别名
- 类继承语句里使用表达式
- Class表达式
- tsconfig.json的 `exclude` 属性
- 用户定义的类型保护函数
- 增强外部模块解析
- JSX支持
- 交叉类型
- `abstract` 类和方法
- 严格的对象字面量赋值检查
- 类和接口的声明合并
- 新增--init

1.5

- 支持解构
- 支持展开操作符
- 支持ES6模块
- 支持for..of
- 支持ES6 Unicode 规范
- 支持Symbols
- 支持计算属性
- 支持tsconfig.json文件
- 支持ES3/ES5的let和const
- 支持ES3/ES5带标记的模版
- 暴露一个新的编辑器接口通过TS Server
- 支持ES7 装饰器提案
- 支持装饰器类型元信息
- 新增--rootDir
- 新增ts.transpile API
- 支持--module umd

- 支持--module system
- 新增--noEmitHelpers
- 新增--inlineSourceMap
- 新增--inlineSources
- 新增--newLine
- 新增--isolatedModules
- 支持新的 `namespace` 关键字
- 支持Visual Studio 2015的tsconfig.json
- 增强Visual Studio 2013的模块字面量高亮

1.4

- 支持联合类型和类型保护
- 新增--noEmitOnError
- 新增--target ES6
- 支持Let and Const
- 支持模块字面量
- Library typings for ES6
- 支持Const enums
- 导出语言服务公共API

1.3

- 为新的编译器重写语言服务
- 支持受保护的成员 in classes
- 支持元组类型

TypeScript 新增内容

TypeScript 2.7

更严格的类属性检查

TypeScript 2.7引入了一个新的控制严格性的标记 `--strictPropertyInitialization` ！

使用这个标记会确保类的每个实例属性都会在构造函数里或使用属性初始化器赋值。 在某种意义上，它会明确地进行从变量到类的实例属性的赋值检查。比如：

```
class C {
    foo: number;
    bar = "hello";
    baz: boolean;
//  ~~~
//  Error! Property 'baz' has no initializer and is not assigned
 directly in the constructor.
    constructor() {
        this.foo = 42;
    }
}
```

上例中， `baz` 从未被赋值，因此TypeScript报错了。 如果我们的本意就是让 `baz` 可以为 `undefined` ，那么应该声明它的类型为 `boolean | undefined` 。

在某些场景下，属性会被间接地初始化（使用辅助方法或依赖注入库）。 这种情况下，你可以在属性上使用显式赋值断言来帮助类型系统识别类型。

```
class C {
    foo!: number;
    // ^
    // Notice this exclamation point!
    // This is the "definite assignment assertion" modifier.
    constructor() {
        this.initialize();
    }

    initialize() {
        this.foo = 0;
    }
}
```

注意， `--strictPropertyInitialization` 会连同其它 `--strict` 模式标记一起被启用，这可能会影响你的工程。 你可以在 `tsconfig.json` 的 `compilerOptions` 里将 `strictPropertyInitialization` 设置为 `false` ， 或者在命令行上将 `--strictPropertyInitialization` 设置为 `false` 来关闭检查。

显式赋值断言

尽管我们尝试将类型系统做的更富表现力，但我们知道有时用户比TypeScript更加了解类型。

上面提到过，显式赋值断言是一个新语法，使用它来告诉TypeScript一个属性会被明确地赋值。 但是除了在类属性上使用它之外，在TypeScript 2.7里你还可以在变量声明上使用它！

```
let x!: number[];
initialize();
x.push(4);

function initialize() {
    x = [0, 1, 2, 3];
}
```

假设我们没有在 `x` 后面加上感叹号，那么TypeScript会报告 `x` 从未被初始化过。它在延迟初始化或重新初始化的场景下很方便使用。

更便利的与ECMAScript模块的互通性

ECMAScript模块在ES2015里才被标准化，在这之前，JavaScript生态系统里存在几种不同的模块 格式，它们工作方式各有不同。 当新的标准通过后，社区遇到了一个难题，就是如何在已有的“老式”模块模式之间保证最佳的互通性。

TypeScript与Babel采取了不同的方案，并且直到现在，还没出现真正地固定标准。简单地说，如果你使用Babel，Webpack或React Native，并期望与你以往使用地不同的导入行为， 我们提供了一个新的编译选项 `--esModuleInterop` 。

TypeScript与Babel都允许用户导入CommonJS模块做为默认导入，但是仍然在导入的命名空间上提 供了每个属性（除非模块使用了 `__esModule` 标记）。

```
import _, { pick } from "lodash";

_.pick(...);
pick(...);
```

由于TypeScript的不同行为，我们在TypeScript 1.8里增加了 `--allowSyntheticDefaultImports` 标记，允许用户在这种行为下检查类型（并非输出）。

通常，在TypeScript视角下的CommonJS（和AMD）模块，命名空间导入总是相当于CommonJS模块对 象的结构，一个默认导入仅相当于模块上一个名字叫做 `default` 的成员。 在这种假定下，你可以创建一个命名导入

```
import { range } from "lodash";

for (let i of range(10)) {
    // ...
}
```

然而，ES命名空间导入不能被调用，因此这种方案并非总是可行。

```
import * as express from "express";

// Should be an error in any valid implementation.
let app = express();
```

为了允许用户使用与Babel或Webpack一致的运行时行为，TypeScript提供了一个新的 `--esModuleInterop` 标记，它用于输出旧模块格式。

当使用这个新的 `--esModuleInterop` 标记时，可调用的CommonJS模块必须被做为默认导入：

```
import express from "express";

let app = express();
```

我们强烈建议Node.js用户利用这个标记，当一个库的模块输出目标为commonjs时，例如express，它会导入一个可调用/可构造的模块。

Webpack用户也可以使用它；然而，你们代码应该将目标设置为 `esnext` 且 `moduleResolution` 策略为 `node` 。使用 `esnext` 模块和 `--esModuleInterop` 等同于启用了 `--allowSyntheticDefaultImports` 。

`unique symbol` 类型和常量名属性

TypeScript 2.7对ECMAScript里的 `symbols` 有了更深入的了解，你可以更灵活地使用它们。

一个需求很大的用例是使用 `symbols` 来声明一个类型良好的属性。 比如，看下面的例子：

```
const Foo = Symbol("Foo");
const Bar = Symbol("Bar");

let x = {
    [Foo]: 100,
    [Bar]: "hello",
};

let a = x[Foo]; // has type 'number'
let b = x[Bar]; // has type 'string'
```

你可以看到，TypeScript可以追踪到 `x` 拥有使用符号 `Foo` 和 `Bar` 声明的属性，因为 `Foo` 和 `Bar` 被声明成常量。TypeScript利用了这一点，让 `Foo` 和 `Bar` 具有了一种新类型： `unique symbols` 。

`unique symbols` 是 `symbols` 的子类型，仅可通过调用 `Symbol()` 或 `Symbol.for()` 或由明确的类型注释生成。它们仅出现在常量声明和只读的静态属性上，并且为了引用一个存在的 `unique symbols` 类型，你必须使用 `typeof` 操作符。每个对 `unique symbols` 的引用都意味着一个完全唯一的声明身份。

```
// Works
declare const Foo: unique symbol;

// Error! 'Bar' isn't a constant.
let Bar: unique symbol = Symbol();

// Works - refers to a unique symbol, but its identity is tied t
o 'Foo'.
let Baz: typeof Foo = Foo;

// Also works.
class C {
    static readonly StaticSymbol: unique symbol = Symbol();
}
```

因为每个 `unique symbols` 都有个完全独立的身份，因此两个 `unique symbols` 类型之前不能赋值和比较。

```
const Foo = Symbol();
const Bar = Symbol();

// Error: can't compare two unique symbols.
if (Foo === Bar) {
    // ...
}
```

另一个可能的用例是使用 `symbols` 做为联合标记。

```
// ./ShapeKind.ts
export const Circle = Symbol("circle");
export const Square = Symbol("square");

// ./ShapeFun.ts
import * as ShapeKind from "./ShapeKind";

interface Circle {
    kind: typeof ShapeKind.Circle;
    radius: number;
}

interface Square {
    kind: typeof ShapeKind.Square;
    sideLength: number;
}

function area(shape: Circle | Square) {
    if (shape.kind === ShapeKind.Circle) {
        // 'shape' has type 'Circle'
        return Math.PI * shape.radius ** 2;
    }
    // 'shape' has type 'Square'
    return shape.sideLength ** 2;
}
```

`--watch` 模式下具有更简洁的输出

在TypeScript的 `--watch` 模式下进行重新编译后会清屏。 这样就更方便阅读最近这次编译的输出信息。

更漂亮的 `--pretty` 输出

TypeScript的 `--pretty` 标记可以让错误信息更易阅读和管理。 我们对这个功能进行了两个主要的改进。 首先， `--pretty` 对文件名，诊段代码和行数添加了颜色（感谢Joshua Goldberg）。 其次，格式化了文件名和位置，以变于在常用的终端里使用Ctrl+Click，Cmd+Click，Alt+Click等来跳转到编译器里的相应位置。

数字分隔符

TypeScript 2.7支持ECMAScript的数字分隔符提案。 这个特性允许用户在数字之间使用下划线（_）来对数字分组（就像使用逗号和点来对数字分组那样）。

```
// Constants
const COULOMB = 8.957_551_787e9; // N-m^2 / C^2
const PLANCK = 6.626_070_040e-34; // J-s
const JENNY = 867_5309; // C-A-L^2
```

这些分隔符对于二进制和十六进制同样有用。

```
let bits = 0b0010_1010;
let routine = 0xC0FFEE_F00D_BED;
let martin = 0xF0_1E_
```

注意，可能有些反常识，JavaScript里的数字表示信用卡和电话号并不适当。 这种情况下使用字符串更好。

固定长度元组

TypeScript 2.6之前， `[number, string, string]` 被当作 `[number, string]` 的子类型。 这是由于TypeScript的结构性类型系统而造成的； `[number, string, string]` 的第一个和第二个元素是 `[number, string]` 里相应的第一个

和第 二个元素的子类型，并且“末尾的”字符串类型是可以赋值给 `[number, string]` 里元素的联合类型的。 然而，在查看了现实中元组的真实用法后，我们注意到大多数情况下这是不被允许的。

感谢Tycho Grouwstra提交的PR，元组类型会考虑它的长度，不同长度的元组不再允许相互赋值。 它通过数字字面量类型实现，因此现在可以区分出不同长度的元组了。

概念上讲，你可以把 `[number, string]` 类型等同于下面的 `NumStrTuple` 声明：

```
interface NumStrTuple extends Array<number | string> {
    0: number;
    1: string;
    length: 2; // using the numeric literal type '2'
}
```

请注意，这是一个破坏性改动。 如果你想要以前的行为，让元组仅限制最小的长度，那么你可以使用一个类似的声明但不明确地指定长 度属性，回退到数字。

```
interface MinimumNumStrTuple extends Array<number | string> {
    0: number;
    1: string;
}
```

`in` 操作符细化和精确的 `instanceof`

TypeScript 2.7带来了两处类型细化方面的改动 - 通过执行“类型保护”确定更详细类型的能力。

首先， `instanceof` 操作符现在利用继承链而非依赖于结构兼容性， 能更准确地反映出 `instanceof` 操作符在运行时的行为。 这可以帮助避免一些复杂的问题，当使用 `instanceof` 去细化结构上相似（但无关）的类型时。

其次，感谢GitHub用户IdeaHunter， `in` 操作符现在做为类型保护使用，会细化掉没有明确声明的属性名。

```
interface A { a: number };
interface B { b: string };

function foo(x: A | B) {
    if ("a" in x) {
        return x.a;
    }
    return x.b;
}
```

更智能的对象字面量推断

在JavaScript里有一种模式，用户会忽略掉一些属性，稍后在使用的时候那些属性的值为 `undefined` 。

```
let foo = someTest ? { value: 42 } : {};
```

在以前TypeScript会查找 `{ value: number }` 和 `{}` 的最佳超类型，结果是 `{}` 。 这从技术角度上讲是正确的，但并不是很有用。

从2.7版本开始，TypeScript会“规范化”每个对象字面量类型记录每个属性， 为每个 `undefined` 类型属性插入一个可选属性，并将它们联合起来。

在上例中， `foo` 的最类型是 `{ value: number } | { value?: undefined }` 。 结合了TypeScript的细化类型，这让我们可以编写更具表达性的代码且TypeScript也可理解。 看另外一个例子：

```
// Has type
//  | { a: boolean, aData: number, b?: undefined }
//  | { b: boolean, bData: string, a?: undefined }
let bar = Math.random() < 0.5 ?
    { a: true, aData: 100 } :
    { b: true, bData: "hello" };

if (bar.b) {
    // TypeScript now knows that 'bar' has the type
    //
    //   '{ b: boolean, bData: string, a?: undefined }'
    //
    // so it knows that 'bData' is available.
    bar.bData.toLowerCase()
}
```

这里，TypeScript可以通过检查 `b` 属性来细化 `bar` 的类型，然后允许我们访问 `bData` 属性。

TypeScript 2.6

严格函数类型

TypeScript 2.6引入了新的类型检查选项，`--strictFunctionTypes`。`--strictFunctionTypes`选项是`--strict`系列选项之一，也就是说`--strict`模式下它默认是启用的。你可以通过在命令行或tsconfig.json中设置`--strictFunctionTypes false`来单独禁用它。

`--strictFunctionTypes`启用时，函数类型参数的检查是抗变（*contravariantly*）而非双变（*bivariantly*）的。关于变体 (variance) 对于函数类型意义的相关背景，请查看协变（covariance）和抗变（contravariance）是什么？。

这一更严格的检查应用于除方法或构造函数声明以外的所有函数类型。方法被专门排除在外是为了确保带泛型的类和接口（如 Array）总体上仍然保持协变。

考虑下面这个 Animal 是 Dog 和 Cat 的父类型的例子：

```
declare let f1: (x: Animal) => void;
declare let f2: (x: Dog) => void;
declare let f3: (x: Cat) => void;
f1 = f2;  // 启用 --strictFunctionTypes 时错误
f2 = f1;  // 正确
f2 = f3;  // 错误
```

第一个赋值语句在默认的类型检查模式中是允许的，但是在严格函数类型模式下会被标记错误。 通俗地讲，默认模式允许这么赋值，因为它可能是合理的，而严格函数类型模式将它标记为错误，因为它不能被证明合理。 任何一种模式中，第三个赋值都是错误的，因为它永远不合理。

用另一种方式来描述这个例子则是，默认类型检查模式中`T`在类型`(x: T) => void`是双变的（也即协变或抗变），但在严格函数类型模式中`T`是抗变的。

例子

```
interface Comparer<T> {
    compare: (a: T, b: T) => number;
}

declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;

animalComparer = dogComparer;  // 错误
dogComparer = animalComparer;  // 正确
```

现在第一个赋值是错误的。更明确地说，`Comparer<T>` 中的 `T` 因为仅在函数类型参数的位置被使用，是抗变的。

另外，注意尽管有的语言（比如C#和Scala）要求变体标注（variance annotations）（`out` / `in` 或 `+` / `-` ），而由于TypeScript的结构化类型系统，它的变体是由泛型中的类型参数的实际使用自然得出的。

注意：

启用 `--strictFunctionTypes` 时，如果 `compare` 被声明为方法，则第一个赋值依然是被允许的。更明确的说，`Comparer<T>` 中的 `T` 因为仅在方法参数的位置被使用所以是双变的。

```
interface Comparer<T> {
    compare(a: T, b: T): number;
}

declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;

animalComparer = dogComparer;  // 正确，因为双变
dogComparer = animalComparer;  // 正确
```

TypeScript 2.6 还改进了与抗变位置相关的类型推导：

```
function combine<T>(...funcs: ((x: ）=> void)[]): (x: T) => void
 {
    return x => {
        for (const f of funcs) f(x);
    }
}

function animalFunc(x: Animal) {}
function dogFunc(x: Dog) {}

let combined = combine(animalFunc，dogFunc);  // (x: Dog) => voi
d
```

这上面所有 `T` 的推断都来自抗变的位置，由此我们得出 `T` 的最普遍子类型。 这与从协变位置推导出的结果恰恰相反，从协变位置我们得出的是最普遍超类型。

缓存模块中的标签模板对象

TypeScript 2.6修复了标签字符串模板的输出，以更好地遵循ECMAScript标准。 根据ECMAScript 标准，每一次获取模板标签的值时，应该将同一个模板字符串数组对象 (同一个 `TemplateStringArray`) 作为第一个参数传递。 在 TypeScript 2.6 之前，每一次生成的都是全新的模板对象。 虽然字符串的内容是一样的，这样的输出会影响通过识别字符串来实现缓存失效的库，比如 lit-html。

例子

```
export function id(x: TemplateStringsArray) {
    return x;
}

export function templateObjectFactory() {
    return id`hello world`;
}

let result = templateObjectFactory() === templateObjectFactory()
; // TS 2.6 为 true
```

编译后的代码：

```
"use strict";
var __makeTemplateObject = (this && this.__makeTemplateObject) |
| function (cooked, raw) {
    if (Object.defineProperty) { Object.defineProperty(cooked, "
raw", { value: raw }); } else { cooked.raw = raw; }
    return cooked;
};

function id(x) {
    return x;
}

var _a;
function templateObjectFactory() {
    return id(_a || (_a = __makeTemplateObject(["hello world"], [
"hello world"])));
}

var result = templateObjectFactory() === templateObjectFactory()
;
```

> 注意：这一改变引入了新的工具函数，`__makeTemplateObject`；如果你在搭配使用`--importHelpers`和`tslib`，需要更新到 1.8 或更高版本。

本地化的命令行诊断消息

TypeScript 2.6 npm包加入了13种语言的诊断消息本地化版本。 命令行中本地化消息会在使用`--locale`选项时显示。

例子

俄语显示的错误消息：

```
c:\ts>tsc --v
Version 2.6.1

c:\ts>tsc --locale ru --pretty c:\test\a.ts

../test/a.ts(1,5): error TS2322: Тип ""string"" не может быть на
значен для типа "number".

1 var x: number = "string";
      ~
```

中文显示的帮助信息：

```
PS C:\ts> tsc --v
Version 2.6.1

PS C:\ts> tsc --locale zh-cn
版本 2.6.1
语法：tsc [选项] [文件 ...]

示例：tsc hello.ts
    tsc --outFile file.js file.ts
    tsc @args.txt

选项：
 -h, --help                      打印此消息。
 --all                           显示所有编译器选项。
 -v, --version                   打印编译器的版本。
 --init                          初始化 TypeScript 项目并创建 tsconfi
g.json 文件。
 -p 文件或目录, --project 文件或目录      编译给定了其配置文件路径或带 "t
sconfig.json" 的文件夹路径的项目。
 --pretty                        使用颜色和上下文风格化错误和消息(实验)
。
 -w, --watch                     监视输入文件。
 -t 版本, --target 版本              指定 ECMAScript 目标版本："ES3"(
默认)、"ES5"、"ES2015"、"ES2016"、"ES2017" 或 "ESNEXT"。
 -m 种类, --module 种类              指定模块代码生成："none"、"commonj
s"、"amd"、"system"、"umd"、"es2015"或 "ESNext"。
```

```
 --lib                             指定要在编译中包括的库文件:
                                     'es5' 'es6' 'es2015' 'es7' 'es2
016' 'es2017' 'esnext' 'dom' 'dom.iterable' 'webworker' 'scripth
ost' 'es2015.core' 'es2015.collection' 'es2015.generator' 'es201
5.iterable' 'es2015.promise' 'es2015.proxy' 'es2015.reflect' 'es
2015.symbol' 'es2015.symbol.wellknown' 'es2016.array.include' 'e
s2017.object' 'es2017.sharedmemory' 'es2017.string' 'es2017.intl'
 'esnext.asynciterable'
 --allowJs                         允许编译 JavaScript 文件。
 --jsx 种类                         指定 JSX 代码生成："preserve"、"re
act-native" 或 "react"。 -d, --declaration             生成相应的
".d.ts" 文件。
 --sourceMap                       生成相应的 ".map" 文件。
 --outFile 文件                     连接输出并将其发出到单个文件。
 --outDir 目录                      将输出结构重定向到目录。
 --removeComments                  请勿将注释发出到输出。
 --noEmit                          请勿发出输出。
 --strict                          启用所有严格类型检查选项。
 --noImplicitAny                   对具有隐式 "any" 类型的表达式和声明引
发错误。
 --strictNullChecks                启用严格的 NULL 检查。
 --strictFunctionTypes             对函数类型启用严格检查。
 --noImplicitThis                  在带隐式"any" 类型的 "this" 表达式上
引发错误。
 --alwaysStrict                    以严格模式进行分析，并为每个源文件发出
"use strict" 指令。
 --noUnusedLocals                  报告未使用的局部变量上的错误。
 --noUnusedParameters              报告未使用的参数上的错误。
 --noImplicitReturns               在函数中的所有代码路径并非都返回值时报告
错误。
 --noFallthroughCasesInSwitch      报告 switch 语句中遇到 fallthrough
情况的错误。
 --types                           要包含在编译中类型声明文件。
 @<文件>                            从文件插入命令行选项和文件。
```

通过 '// @ts-ignore' 注释隐藏 .ts 文件中的错误

TypeScript 2.6支持在.ts文件中通过在报错一行上方使用 `// @ts-ignore` 来忽略错误。

例子

```
if (false) {
    // @ts-ignore：无法被执行的代码的错误
    console.log("hello");
}
```

`// @ts-ignore` 注释会忽略下一行中产生的所有错误。 建议实践中在 `@ts-ignore` 之后添加相关提示，解释忽略了什么错误。

请注意，这个注释仅会隐藏报错，并且我们建议你极少使用这一注释。

更快的 `tsc --watch`

TypeScript 2.6 带来了更快的 `--watch` 实现。 新版本优化了使用ES模块的代码的生成和检查。 在一个模块文件中检测到的改变只会使改变的模块，以及依赖它的文件被重新生成，而不再是整个项目。 有大量文件的项目应该从这一改变中获益最多。

这一新的实现也为tsserver中的监听带来了性能提升。 监听逻辑被完全重写以更快响应改变事件。

只写的引用现在会被标记未使用

TypeScript 2.6加入了修正的 `--noUnusedLocals` 和 `--noUnusedParameters` 编译选项实现。 只被写但从没有被读的声明现在会被标记未使用。

例子

下面 `n` 和 `m` 都会被标记为未使用，因为它们的值从未被读取。之前 TypeScript 只会检查它们的值是否被引用。

```
function f(n: number) {
    n = 0;
}

class C {
    private m: number;
    constructor() {
        this.m = 0;
    }
}
```

另外仅被自己内部调用的函数也会被认为是未使用的。

例子

```
function f() {
    f(); // 错误：'f' 被声明，但它的值从未被使用
}
```

TypeScript 2.5

可选的 `catch` 语句变量

得益于@tinganho所做的工作，TypeScript 2.5实现了一个新的ECMAScript特性，允许用户省略 `catch` 语句中的变量。 例如，当使用 `JSON.parse` 时，你可能需要将对应的函数调用放在 `try` / `catch` 中，但是最后可能并不会用到输入有误时会抛出的 `SyntaxError` （语法错误）。

```
let input = "...";
try {
    JSON.parse(input);
}
catch {
    // ^ 注意我们的 `catch` 语句并没有声明一个变量
    console.log("传入的 JSON 不合法\n\n" + input)
}
```

`checkJs` / `@ts-check` 模式中的类型断言/转换语法

TypeScript 2.5 引入了在使用纯 JavaScript 的项目中断言表达式类型的能力。对应的语法是 `/** @type {...} */` 标注注释后加上被圆括号括起来，类型需要被重新演算的表达式。举例:

```
var x = /** @type {SomeType} */ (AnyParenthesizedExpression);
```

包去重和重定向

在 TypeScript 2.5 中使用 `Node` 模块解析策略进行导入时，编译器现在会检查文件是否来自 "相同" 的包。如果一个文件所在的包的 `package.json` 包含了与之前读取的包相同的 `name` 和 `version` ，那么TypeScript会将它重定向到最顶层的包。

这可以解决两个包可能会包含相同的类声明，但因为包含 `private` 成员导致他们在结构上不兼容的问题.

这也带来一个额外的好处，可以通过避免从重复的包中加载 `.d.ts` 文件减少内存使用和编译器及语言服务的运行时计算.

`--preserveSymlinks`（保留符号链接）编译器选项

TypeScript 2.5带来了 `preserveSymlinks` 选项，它对应了Node.js 中 `--preserve-symlinks` 选项的行为。这一选项也会带来和Webpack的 `resolve.symlinks` 选项相反的行为（也就是说，将TypeScript的 `preserveSymlinks` 选项设置为 `true` 对应了将Webpack的 `resolve.symlinks` 选项设为 `false` ，反之亦然）。

在这一模式中，对于模块和包的引用（比如 `import` 语句和 `/// <reference type=".." />` 指令）都会以相对符号链接文件的位置被解析，而不是相对于符号链接解析到的路径。更具体的例子，可以参考Node.js网站的文档。

TypeScript 2.4

动态导入表达式

动态的 `import` 表达式是一个新特性，它属于ECMAScript的一部分，允许用户在程序的任何位置异步地请求某个模块。

这意味着你可以有条件地延迟加载其它模块和库。例如下面这个 `async` 函数，它仅在需要的时候才导入工具库：

```
async function getZipFile(name: string, files: File[]): Promise<
File> {
    const zipUtil = await import('./utils/create-zip-file');
    const zipContents = await zipUtil.getContentAsBlob(files);
    return new File(zipContents, name);
}
```

许多bundlers工具已经支持依照这些 `import` 表达式自动地分割输出，因此可以考虑使用这个新特性并把输出模块目标设置为 `esnext` 。

字符串枚举

TypeScript 2.4现在支持枚举成员变量包含字符串构造器。

```
enum Colors {
    Red = "RED",
    Green = "GREEN",
    Blue = "BLUE",
}
```

需要注意的是字符串枚举成员不能被反向映射到枚举成员的名字。换句话说，你不能使用 `Colors["RED"]` 来得到 `"Red"` 。

增强的泛型推断

TypeScript 2.4围绕着泛型的推断方式引入了一些很棒的改变。

返回类型作为推断目标

其一，TypeScript能够推断调用的返回值类型。 这可以优化你的体验和方便捕获错误。 如下所示：

```
function arrayMap<T, U>(f: (x: T) => U): (a: T[]) => U[] {
    return a => a.map(f);
}

const lengths: (a: string[]) => number[] = arrayMap(s => s.length);
```

下面是一个你可能会见到的出错了的例子：

```
let x: Promise<string> = new Promise(resolve => {
    resolve(10);
    //      ~~ Error!
});
```

从上下文类型中推断类型参数

在TypeScript 2.4之前，在下面的例子里：

```
let f: <T>(x: T) => T = y => y;
```

`y` 将会具有 `any` 类型。 这意味着虽然程序会检查类型，但是你却可以使用 `y` 做任何事情，就比如：

```
let f: <T>(x: T) => T = y => y() + y.foo.bar;
```

这个例子实际上并不是类型安全的。

在TypeScript 2.4里，右手边的函数会隐式地获得类型参数，并且 `y` 的类型会被推断为那个类型参数的类型。

如果你使用 `y` 的方式是这个类型参数所不支持的，那么你会得到一个错误。 在这个例子里， `T` 的约束是 `{}` （隐式地），所以在最后一个例子里会出错。

对泛型函数进行更严格的检查

TypeScript在比较两个单一签名的类型时会尝试统一类型参数。 因此，在涉及到两个泛型签名的时候会进行更严格的检查，这就可能发现一些bugs。

```
type A = <T, U>(x: T, y: U) => [T, U];
type B = <S>(x: S, y: S) => [S, S];

function f(a: A, b: B) {
    a = b;  // Error
    b = a;  // Ok
}
```

回调参数的严格抗变

TypeScript一直是以双变（bivariant）的方式来比较参数。 这样做有很多原因，总体上来说这不会有什么大问题直到我们发现它应用在 `Promise` 和 `Observable` 上时有些副作用。

TypeScript 2.4在处理两个回调类型时引入了收紧机制。例如：

```
interface Mappable<T> {
    map<U>(f: (x: T) => U): Mappable<U>;
}

declare let a: Mappable<number>;
declare let b: Mappable<string | number>;

a = b;
b = a;
```

在TypeScript 2.4之前，它会成功执行。 当关联 `map` 的类型时，TypeScript会双向地关联它们的类型（例如 `f` 的类型）。 当关联每个 `f` 的类型时，TypeScript也会双向地关联那些参数的类型。

TS 2.4里关联 `map` 的类型时，TypeScript会检查是否每个参数都是回调类型，如果是的话，它会确保那些参数根据它所在的位置以抗变（contravariant）地方式进行检查。

换句话说，TypeScript现在可以捕获上面的bug，这对某些用户来说可能是一个破坏性改动，但却是非常帮助的。

弱类型（Weak Type）探测

TypeScript 2.4引入了“弱类型”的概念。 任何只包含了可选属性的类型被当作是“weak”。 比如，下面的 `Options` 类型是弱类型：

```
interface Options {
    data?: string,
    timeout?: number,
    maxRetries?: number,
}
```

在TypeScript 2.4里给弱类型赋值时，如果这个值的属性与弱类型的属性没有任何重叠属性时会得到一个错误。 比如：

```
function sendMessage(options: Options) {
    // ...
}

const opts = {
    payload: "hello world!",
    retryOnFail: true,
}

// 错误!
sendMessage(opts);
// 'opts' 和 'Options' 没有重叠的属性
// 可能我们想要用'data'/'maxRetries'来代替'payload'/'retryOnFail'
```

因为这是一个破坏性改动，你可能想要知道一些解决方法：

1. 确定属性存在时再声明

2. 给弱类型增加索引签名（比如 `[propName: string]: {}` ）
3. 使用类型断言（比如 `opts as Options` ）

TypeScript 2.3

ES5/ES3 的生成器和迭代支持

首先是一些 *ES2016* 的术语：

迭代器

ES2015引入了 `Iterator`（迭代器），它表示提供了 next，return，以及 throw 三个方法的对象，具体满足以下接口：

```
interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}
```

这种迭代器对于迭代可用的值时很有用，比如数组的元素或者Map的键。如果一个对象有一个返回 `Iterator` 对象的 `Symbol.iterator` 方法，那么我们说这个对象是“可迭代的”。

迭代器协议还定义了一些ES2015中的特性像 `for..of` 和展开运算符以及解构赋值中的数组的剩余运算的操作对象。

生成器

ES2015也引入了"生成器"，生成器是可以通过 `Iterator` 接口和 `yield` 关键字被用来生成部分运算结果的函数。生成器也可以在内部通过 `yield*` 代理对与其他可迭代对象的调用。举例来说：

```
function* f() {
  yield 1;
  yield* [2, 3];
}
```

新的 **`--downlevelIteration`** 编译选项

之前迭代器只在编译目标为 ES6/ES2015 或者更新版本时可用。此外，设计迭代器协议的结构，比如 `for..of` ，如果编译目标低于ES6/ES2015，则只能在操作数组时被支持。

TypeScript 2.3 在 ES3 和 ES5 为编译目标时由 `--downlevelIteration` 编译选项增加了完整的对生成器和迭代器协议的支持。

通过 `--downlevelIteration` 编译选项，编译器会使用新的类型检查和输出行为，尝试调用被迭代对象的 `[Symbol.iterator]()` 方法 (如果有)，或者在对象上创建一个语义上的数组迭代器。

> 注意这需要非数组的值有原生的 `Symbol.iterator` 或者 `Symbol.iterator` 的运行时模拟实现。

使用 `--downlevelIteration` 时，在 ES5/ES3 中 `for..of` 语句、数组解构、数组中的元素展开、函数调用、new 表达式在支持 `Symbol.iterator` 时可用，但即便没有定义 `Symbol.iterator` ，它们在运行时或开发时都可以被使用到数组上.

异步迭代

TypeScript 2.3 添加了对异步迭代器和生成器的支持，描述见当前的TC39 提案。

异步迭代器

异步迭代引入了 `AsyncIterator` ，它和 `Iterator` 相似。实际上的区别在于 `AsyncIterator` 的 `next` 、 `return` 和 `throw` 方法的返回的是迭代结果的 `Promise` ，而不是结果本身。这允许 `AsyncIterator` 在生成值之前的时间点就加入异步通知。 `AsyncIterator` 的接口如下：

```
interface AsyncIterator<T> {
  next(value?: any): Promise<IteratorResult<T>>;
  return?(value?: any): Promise<IteratorResult<T>>;
  throw?(e?: any): Promise<IteratorResult<T>>;
}
```

一个支持异步迭代的对象如果有一个返回 `AsyncIterator` 对象的 `Symbol.asyncIterator` 方法，被称作是“可迭代的”。

异步生成器

异步迭代提案引入了“异步生成器”，也就是可以用来生成部分计算结果的异步函数。异步生成器也可以通过 `yield*` 代理对可迭代对象或异步可迭代对象的调用：

```
async function* g() {
  yield 1;
  await sleep(100);
  yield* [2, 3];
  yield* (async function *() {
    await sleep(100);
    yield 4;
  })();
}
```

和生成器一样，异步生成器只能是函数声明，函数表达式，或者类或对象字面量的方法。箭头函数不能作为异步生成器。异步生成器除了一个可用的 `Symbol.asyncIterator` 引用外(原生或三方实现)，还需要一个可用的全局 `Promise` 实现（既可以是原生的，也可以是ES2015兼容的实现）。

`for-await-of` 语句

最后，ES2015引入了 `for..of` 语句来迭代可迭代对象。相似的，异步迭代提案引入了 `for..await..of` 语句来迭代可异步迭代的对象。

```
async function f() {
  for await (const x of g()) {
     console.log(x);
  }
}
```

`for..await..of` 语句仅在异步函数或异步生成器中可用。

注意事项

- 始终记住我们对于异步迭代器的支持是建立在运行时有 `Symbol.asyncIterator` 支持的基础上的。你可能需要 `Symbol.asyncIterator` 的三方实现，虽然对于简单的目的可以仅仅

是： `(Symbol as any).asyncIterator = Symbol.asyncIterator || Symbol.for("Symbol.asyncIterator");`
- 如果你没有声明 `AsyncIterator` ，还需要在 `--lib` 选项中加入 `esnext` 来获取 `AsyncIterator` 声明。
- 最后, 如果你的编译目标是ES5或ES3，你还需要设置 `--downlevelIterators` 编译选项。

泛型参数默认类型

TypeScript 2.3 增加了对声明泛型参数默认类型的支持。

示例

考虑一个会创建新的 `HTMLElement` 的函数，调用时不加参数会生成一个 `Div` ，你也可以选择性地传入子元素的列表。之前你必须这么去定义：

```
declare function create(): Container<HTMLDivElement, HTMLDivElement[]>;
declare function create<T extends HTMLElement>(element: T): Container<T, T[]>;
declare function create<T extends HTMLElement, U extends HTMLElement>(element: T, children: U[]): Container<T, U[]>;
```

有了泛型参数默认类型，我们可以将定义化简为：

```
declare function create<T extends HTMLElement = HTMLDivElement, U = T[]>(element?: T, children?: U): Container<T, U>;
```

泛型参数的默认类型遵循以下规则：

- 有默认类型的类型参数被认为是可选的。
- 必选的类型参数不能在可选的类型参数后。
- 如果类型参数有约束，类型参数的默认类型必须满足这个约束。
- 当指定类型实参时，你只需要指定必选类型参数的类型实参。 未指定的类型参数会被解析为它们的默认类型。
- 如果指定了默认类型，且类型推断无法选择一个候选类型，那么将使用默认类型作为推断结果。

- 一个被现有类或接口合并的类或者接口的声明可以为现有类型参数引入默认类型。
- 一个被现有类或接口合并的类或者接口的声明可以引入新的类型参数，只要它指定了默认类型。

新的 `--strict` 主要编译选项

TypeScript加入的新检查项为了避免不兼容现有项目通常都是默认关闭的。虽然避免不兼容是好事，但这个策略的一个弊端则是使配置最高类型安全越来越复杂，这么做每次TypeScript版本发布时都需要显示地加入新选项。有了 `--strict` 编译选项，就可以选择最高级别的类型安全（了解随着更新版本的编译器增加了增强的类型检查特性可能会报新的错误）。

新的 `--strict` 编译器选项包含了一些建议配置的类型检查选项。具体来说，指定 `--strict` 相当于是指定了以下所有选项（未来还可能包括更多选项）：

- `--strictNullChecks`
- `--noImplicitAny`
- `--noImplicitThis`
- `--alwaysStrict`

确切地说，`--strict` 编译选项会为以上列出的编译器选项设置默认值。这意味着还可以单独控制这些选项。比如：

```
--strict --noImplicitThis false
```

这将是开启除 `--noImplicitThis` 编译选项以外的所有严格检查选项。使用这个方式可以表述除某些明确列出的项以外的所有严格检查项。换句话说，现在可以在默认最高级别的类型安全下排除部分检查。

从TypeScript 2.3开始，`tsc --init` 生成的默认 `tsconfig.json` 在 `"compilerOptions"` 中包含了 `"strict: true"` 设置。这样一来，用 `tsc --init` 创建的新项目默认会开启最高级别的类型安全。

改进的 `--init` 输出

除了默认的 `--strict` 设置外， `tsc --init` 还改进了输出。 `tsc --init` 默认生成的 `tsconfig.json` 文件现在包含了一些带描述的被注释掉的常用编译器选项.你可以去掉相关选项的注释来获得期望的结果。我们希望新的输出能简化新项目的配置并且随着项目成长保持配置文件的可读性。

`--checkJS` 选项下 .js 文件中的错误

即便使用了 `--allowJs` ，TypeScript编译器默认不会报 .js 文件中的任何错误。TypeScript 2.3 中使用 `--checkJs` 选项， `.js` 文件中的类型检查错误也可以被报出.

你可以通过为它们添加 `// @ts-nocheck` 注释来跳过对某些文件的检查，反过来你也可以选择通过添加 `// @ts-check` 注释只检查一些 `.js` 文件而不需要设置 `--checkJs` 编译选项。你也可以通过添加 `// @ts-ignore` 到特定行的一行前来忽略这一行的错误.

`.js` 文件仍然会被检查确保只有标准的 ECMAScript 特性，类型标注仅在 `.ts` 文件中被允许，在 `.js` 中会被标记为错误。JSDoc注释可以用来为你的JavaScript代码添加某些类型信息，更多关于支持的JSDoc结构的详情，请浏览JSDoc支持文档。

有关详细信息，请浏览类型检查JavaScript文件文档。

TypeScript 2.2

支持混合类

TypeScript 2.2 增加了对 ECMAScript 2015 混合类模式 (见MDN混合类的描述及JavaScript类的"真"混合了解更多) 以及使用交叉来类型表达结合混合构造函数的签名及常规构造函数签名的规则.

首先是一些术语

混合构造函数类型指仅有单个构造函数签名，且该签名仅有一个类型为 any[] 的变长参数，返回值为对象类型. 比如, 有 X 为对象类型, new (...args: any[]) => X 是一个实例类型为 X 的混合构造函数类型。

混合类指一个 `extends` (扩展)了类型参数类型的表达式的类声明或表达式. 以下规则对混合类声明适用：

- `extends` 表达式的类型参数类型必须是混合构造函数.
- 混合类的构造函数 (如果有) 必须有且仅有一个类型为 `any[]` 的变长参数, 并且必须使用展开运算符在 `super(...args)` 调用中将这些参数传递。

假设有类型参数为 `T` 且约束为 `X` 的表达式 `Bas` ，处理混合类 `class C extends Base {...}` 时会假设 `Base` 有 `X` 类型，处理结果为交叉类型 `typeof C & T` 。换言之，一个混合类被表达为混合类构造函数类型与参数基类构造函数类型的交叉类型.

在获取一个包含了混合构造函数类型的交叉类型的构造函数签名时，混合构造函数签名会被丢弃，而它们的实例类型会被混合到交叉类型中其他构造函数签名的返回类型中. 比如，交叉类型 `{ new(...args: any[]) => A } & { new(s: string) => B }` 仅有一个构造函数签名 `new(s: string) => A & B` 。

将以上规则放到一个例子中

```
class Point {
    constructor(public x: number, public y: number) {}
}

class Person {
    constructor(public name: string) {}
}

type Constructor<T> = new(...args: any[]) => T;

function Tagged<T extends Constructor<{}>>(Base: T) {
    return class extends Base {
        _tag: string;
        constructor(...args: any[]) {
            super(...args);
            this._tag = "";
        }
    }
}

const TaggedPoint = Tagged(Point);

let point = new TaggedPoint(10, 20);
point._tag = "hello";

class Customer extends Tagged(Person) {
    accountBalance: number;
}

let customer = new Customer("Joe");
customer._tag = "test";
customer.accountBalance = 0;
```

混合类可以通过在类型参数中限定构造函数签名的返回值类型来限制它们可以被混入的类的类型。举例来说，下面的 `WithLocation` 函数实现了一个为满足 `Point` 接口（也就是有类型为 `number` 的 `x` 和 `y` 属性）的类添加 `getLocation` 方法的子类工厂。

```
interface Point {
    x: number;
    y: number;
}

const WithLocation = <T extends Constructor<Point>>(Base: T) =>
    class extends Base {
        getLocation(): [number, number] {
            return [this.x, this.y];
        }
    }
```

`object` 类型

TypeScript没有表示非基本类型的类型，即不是 `number` | `string` | `boolean` | `symbol` | `null` | `undefined` 的类型。一个新的 `object` 类型登场。

使用 `object` 类型，可以更好地表示类似 `Object.create` 这样的API。例如：

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

支持 `new.target`

`new.target` 元属性是ES2015引入的新语法。当通过 `new` 构造函数创建实例时， `new.target` 的值被设置为对最初用于分配实例的构造函数的引用。如果一个函数不是通过 `new` 构造而是直接被调用，那么 `new.target` 的值被设置为 `undefined` 。

当在类的构造函数中需要设置 `Object.setPrototypeOf` 或 `__proto__` 时， `new.target` 就派上用场了。在NodeJS v4及更高版本中继承 `Error` 类就是这样的使用案例。

示例

```
class CustomError extends Error {
    constructor(message?: string) {
        super(message); // 'Error' breaks prototype chain here
        Object.setPrototypeOf(this, new.target.prototype); // restore prototype chain
    }
}
```

生成JS代码：

```
var CustomError = (function (_super) {
  __extends(CustomError, _super);
  function CustomError() {
    var _newTarget = this.constructor;
    var _this = _super.apply(this, arguments);  // 'Error' breaks prototype chain here
    _this.__proto__ = _newTarget.prototype; // restore prototype chain
    return _this;
  }
  return CustomError;
})(Error);
```

new.target也适用于编写可构造的函数，例如：

```
function f() {
  if (new.target) { /* called via 'new' */ }
}
```

编译为：

```
function f() {
  var _newTarget = this && this instanceof f ? this.constructor
: void 0;
  if (_newTarget) { /* called via 'new' */ }
}
```

更好地检查表达式的操作数中的 `null` / `undefined`

TypeScript 2.2改进了对表达式中可空操作数的检查。具体来说，这些现在被标记为错误：

- 如果 `+` 运算符的任何一个操作数是可空的，并且两个操作数都不是 `any` 或 `string` 类型。
- 如果 `-` ， `*` ， `**` ， `/` ， `%` ， `<<` ， `>>` ， `>>>` , `&` , `|` 或 `^` 运算符的任何一个操作数是可空的。
- 如果 `<` ， `>` ， `<=` ， `>=` 或 `in` 运算符的任何一个操作数是可空的。
- 如果 `instanceof` 运算符的右操作数是可空的。
- 如果一元运算符 `+` ， `-` ， `~` ， `++` 或者 `--` 的操作数是可空的。

如果操作数的类型是 `null` 或 `undefined` 或者包含 `null` 或 `undefined` 的联合类型，则操作数视为可空的。注意：包含 `null` 或 `undefined` 的联合类型只会出现在 `--strictNullChecks` 模式中，因为常规类型检查模式下 `null` 和 `undefined` 在联合类型中是不存在的。

字符串索引签名类型的点属性

具有字符串索引签名的类型可以使用 `[]` 符号访问，但不允许使用 `.` 符号访问。从TypeScript 2.2开始两种方式都允许使用。

```
interface StringMap<T> {
    [x: string]: T;
}

const map: StringMap<number>;

map["prop1"] = 1;
map.prop2 = 2;
```

这仅适用于具有显式字符串索引签名的类型。在类型使用上使用 `.` 符号访问未知属性仍然是一个错误。

支持在JSX子元素上使用扩展运算符

TypeScript 2.2增加了对在JSX子元素上使用扩展运算符的支持。更多详情请看facebook/jsx#57。

示例

```
function Todo(prop: { key: number, todo: string }) {
    return <div>{prop.key.toString() + prop.todo}</div>;
}

function TodoList({ todos }: TodoListProps) {
    return <div>
        {...todos.map(todo => <Todo key={todo.id} todo={todo.tod
o} />)}
    </div>;
}

let x: TodoListProps;

<TodoList {...x} />
```

新的 `jsx: react-native`

React-native构建管道期望所有文件都具有.js扩展名，即使该文件包含JSX语法。新的 `--jsx` 编译参数值 `react-native` 将在输出文件中坚持JSX语法，但是给它一个 `.js` 扩展名。

TypeScript 2.1

`keyof` 和查找类型

在JavaScript中属性名称作为参数的API是相当普遍的，但是到目前为止还没有表达在那些API中出现的类型关系。

输入索引类型查询或 `keyof`，索引类型查询 `keyof T` 产生的类型是 `T` 的属性名称。 `keyof T` 的类型被认为是 `string` 的子类型。

示例

```
interface Person {
    name: string;
    age: number;
    location: string;
}

type K1 = keyof Person; // "name" | "age" | "location"
type K2 = keyof Person[];  // "length" | "push" | "pop" | "conca
t" | ...
type K3 = keyof { [x: string]: Person };  // string
```

与之相对应的是索引访问类型，也称为查找类型。在语法上，它们看起来像元素访问，但是写成类型：

示例

```
type P1 = Person["name"];  // string
type P2 = Person["name" | "age"];  // string | number
type P3 = string["charAt"];  // (pos: number) => string
type P4 = string[]["push"];  // (...items: string[]) => number
type P5 = string[][0];  // string
```

你可以将这种模式和类型系统的其它部分一起使用，以获取类型安全的查找。

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];  // 推断类型是T[K]
}

function setProperty<T, K extends keyof T>(obj: T, key: K, value
: T[K]) {
    obj[key] = value;
}

let x = { foo: 10, bar: "hello!" };

let foo = getProperty(x, "foo"); // number
let bar = getProperty(x, "bar"); // string

let oops = getProperty(x, "wargarbl"); // 错误！"wargarbl"不存在"f
oo" | "bar"中

setProperty(x, "foo", "string"); // 错误！, 类型是number而非string
```

映射类型

一个常见的任务是使用现有类型并使其每个属性完全可选。假设我们有一个 `Person` ：

```
interface Person {
    name: string;
    age: number;
    location: string;
}
```

`Person` 的可选属性类型将是这样：

```
interface PartialPerson {
    name?: string;
    age?: number;
    location?: string;
}
```

使用映射类型， `PartialPerson` 可以写成是 `Person` 类型的广义变换：

```
type Partial<T> = {
    [P in keyof T]?: T[P];
};

type PartialPerson = Partial<Person>;
```

映射类型是通过使用字面量类型的集合而生成的，并为新对象类型计算一组属性。它们就像Python中的列表推导式，但不是在列表中产生新的元素，而是在类型中产生新的属性。

除 `Partial` 外，映射类型可以表示许多有用的类型转换：

```
// 保持类型相同，但每个属性是只读的。
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

// 相同的属性名称，但使值是一个Promise，而不是一个具体的值
type Deferred<T> = {
    [P in keyof T]: Promise<T[P]>;
};

// 为T的属性添加代理
type Proxify<T> = {
    [P in keyof T]: { get(): T[P]; set(v: T[P]): void }
};
```

`Partial` , `Readonly` , `Record` 和 `Pick`

`Partial` 和 `Readonly` ，如前所述，是非常有用的结构。你可以使用它们来描述像一些常见的JS程序：

```
function assign<T>(obj: T, props: Partial<T>): void;
function freeze<T>(obj: T): Readonly<T>;
```

因此，它们现在默认包含在标准库中。

我们还包括两个其他实用程序类型： `Record` 和 `Pick` 。

```
// 从T中选取一组属性K
declare function pick<T, K extends keyof T>(obj: T, ...keys: K[]
): Pick<T, K>;

const nameAndAgeOnly = pick(person, "name", "age");  // { name:
string, age: number }
```

```
// 对于类型T的每个属性K，将其转换为U
function mapObject<K extends string | number, T, U>(obj: Record<
K, T>, f: (x: T) => U): Record<K, U>

const names = { foo: "hello", bar: "world", baz: "bye" };
const lengths = mapObject(names, s => s.length);  // { foo: numb
er, bar: number, baz: number }
```

对象扩展运算符和rest运算符

TypeScript 2.1带来了ESnext扩展运算符和rest运算符的支持。

类似于数组扩展，展开对象可以方便得到浅拷贝：

```
let copy = { ...original };
```

同样，您可以合并几个不同的对象。在以下示例中，合并将具有来自 `foo` ， `bar` 和 `baz` 的属性。

```
let merged = { ...foo, ...bar, ...baz };
```

还可以重写现有属性并添加新属性.：

```
let obj = { x: 1, y: "string" };
var newObj = {...obj, z: 3, y: 4}; // { x: number, y: number, z:
 number }
```

指定展开操作的顺序确定哪些属性在最终的结果对象中。相同的属性，后面的属性会“覆盖”前面的属性。

与对象扩展运算符相对的是对象rest运算符，因为它可以提取解构元素中剩余的元素：

```
let obj = { x: 1, y: 1, z: 1 };
let { z, ...obj1 } = obj;
obj1; // {x: number, y: number};
```

低版本异步函数

该特性在TypeScript 2.1之前就已经支持了，但是只能编译为ES6或者ES2015。TypeScript 2.1使其该特性可以在ES3和ES5运行时上使用，这意味着无论您使用什么环境，都可以使用它。

> 注：首先，我们需要确保我们的运行时提供全局的ECMAScript兼容性 `Promise` 。这可能需要获取 `Promise` 的polyfill，或者依赖运行时的版本。我们还需要通过设置 `lib` 编译参数，比如 `"dom","es2015"` 或 `"dom","es2015.promise","es5"` 来确保TypeScript知道 `Promise` 可用。

示例

tsconfig.json

```
{
    "compilerOptions": {
        "lib": ["dom", "es2015.promise", "es5"]
    }
}
```

dramaticWelcome.ts

```
function delay(milliseconds: number) {
    return new Promise<void>(resolve => {
        setTimeout(resolve, milliseconds);
    });
}

async function dramaticWelcome() {
    console.log("Hello");

    for (let i = 0; i < 3; i++) {
        await delay(500);
        console.log(".");
    }

    console.log("World!");
}

dramaticWelcome();
```

编译和运行输出应该会在ES3/ES5引擎上产生正确的行为。

支持外部辅助库（`tslib`）

TypeScript注入了一些辅助函数，如继承 `_extends` 、JSX中的展开运算符 `__assign` 和异步函数 `__awaiter` 。

以前有两个选择：

1. 在每一个需要辅助库的文件都注入辅助库或者
2. 使用 `--noEmitHelpers` 编译参数完全不使用辅助库。

这两项还有待改进。将帮助文件捆绑在每个文件中对于试图保持其包尺寸小的客户而言是一个痛点。不使用辅助库，那么客户就必须自己维护辅助库。

TypeScript 2.1 允许这些辅助库作为单独的模块一次性添加到项目中，并且编译器根据需求导入它们。

首先，安装 `tslib` ：

```
npm install tslib
```

然后，使用 `--importHelpers` 编译你的文件：

```
tsc --module commonjs --importHelpers a.ts
```

因此下面的输入，生成的 `.js` 文件将包含 `tslib` 的导入和使用 `__assign` 辅助函数替代内联操作。

```
export const o = { a: 1, name: "o" };
export const copy = { ...o };
```

```
"use strict";
var tslib_1 = require("tslib");
exports.o = { a: 1, name: "o" };
exports.copy = tslib_1.__assign({}, exports.o);
```

无类型导入

TypeScript历来对于如何导入模块过于严格。这是为了避免输入错误，并防止用户错误地使用模块。

但是，很多时候你可能只想导入的现有模块，但是这些模块可能没有 `.d.ts` 文件。以前这是错误的。从TypeScript 2.1开始，这更容易了。

使用TypeScript 2.1，您可以导入JavaScript模块，而不需要类型声明。如果类型声明（如 `declare module "foo" { ... }` 或 `node_modules/@types/foo` ）存在，则仍然优先。

对于没有声明文件的模块的导入，在使用了 `--noImplicitAny` 编译参数后仍将被标记为错误。

```
// Succeeds if `node_modules/asdf/index.js` exists
import { x } from "asdf";
```

支持 `--target ES2016` , `--target ES2017` 和 `--target ESNext`

TypeScript 2.1支持三个新的编译版本值 `--target ES2016` , `--target ES2017` 和 `--target ESNext` 。

使用target `--target ES2016` 将指示编译器不要编译ES2016特有的特性，比如 `**` 操作符。

同样， `--target ES2017` 将指示编译器不要编译ES2017特有的特性像 `async/await` 。

`--target ESNext` 则对应最新的ES提议特性支持.

改进 `any` 类型推断

以前，如果TypeScript无法确定变量的类型，它将选择 `any` 类型。

```
let x;      // 隐式 'any'
let y = []; // 隐式 'any[]'

let z: any; // 显式 'any'.
```

使用TypeScript 2.1，TypeScript不是仅仅选择 `any` 类型，而是基于你后面的赋值来推断类型。

仅当设置了 `--noImplicitAny` 编译参数时，才会启用此选项。

示例

```
let x;

// 你仍然可以给'x'赋值任何你需要的任何值。
x = () => 42;

// 在刚赋值后，TypeScript 2.1 知道'x'的类型是'() => number'。
let y = x();

// 感谢，现在它会告诉你，你不能添加一个数字到一个函数！
console.log(x + y);
//          ~~~~~
// 错误！运算符 '+' 不能应用于类型`() => number`和'number'。

// TypeScript仍然允许你给'x'赋值你需要的任何值。
x = "Hello world!";

// 并且现在它也知道'x'是'string'类型的！
x.toLowerCase();
```

现在对空数组也进行同样的跟踪。

没有类型注解并且初始值为 `[]` 的变量被认为是一个隐式的 `any[]` 变量。变量会根据下面这些操作 `x.push(value)` 、 `x.unshift(value)` 或 `x[n] = value` 向其中添加的元素来不断改变自身的类型。

```
function f1() {
    let x = [];
    x.push(5);
    x[1] = "hello";
    x.unshift(true);
    return x;  // (string | number | boolean)[]
}

function f2() {
    let x = null;
    if (cond()) {
        x = [];
        while (cond()) {
            x.push("hello");
        }
    }
    return x;  // string[] | null
}
```

隐式any错误

这样做的一个很大的好处是，当使用 `--noImplicitAny` 运行时，你将看到较少的隐式 `any` 错误。隐式 `any` 错误只会在编译器无法知道一个没有类型注解的变量的类型时才会报告。

示例

```
function f3() {
    let x = [];  // 错误：当变量'x'类型无法确定时，它隐式具有'any[]'类型。
    x.push(5);
    function g() {
        x;    // 错误：变量'x'隐式具有'any【】'类型。
    }
}
```

更好的字面量类型推断

字符串、数字和布尔字面量类型（如：`"abc"`，`1`和`true`）之前仅在存在显式类型注释时才被推断。从TypeScript 2.1开始，字面量类型总是推断为默认值。

不带类型注解的`const`变量或`readonly`属性的类型推断为字面量初始化的类型。已经初始化且不带类型注解的`let`变量、`var`变量、形参或非`readonly`属性的类型推断为初始值的扩展字面量类型。字符串字面量扩展类型是`string`，数字字面量扩展类型是`number`，`true`或`false`的字面量类型是`boolean`，还有枚举字面量扩展类型是枚举。

示例

```
const c1 = 1;  // Type 1
const c2 = c1;  // Type 1
const c3 = "abc";  // Type "abc"
const c4 = true;  // Type true
const c5 = cond ? 1 : "abc";  // Type 1 | "abc"

let v1 = 1;  // Type number
let v2 = c2;  // Type number
let v3 = c3;  // Type string
let v4 = c4;  // Type boolean
let v5 = c5;  // Type number | string
```

字面量类型扩展可以通过显式类型注解来控制。具体来说，当为不带类型注解的`const`局部变量推断字面量类型的表达式时，`var`变量获得扩展字面量类型推断。但是当`const`局部变量有显式字面量类型注解时，`var`变量获得非扩展字面量类型。

示例

```
const c1 = "hello";  // Widening type "hello"
let v1 = c1;  // Type string

const c2: "hello" = "hello";  // Type "hello"
let v2 = c2;  // Type "hello"
```

将基类构造函数的返回值作为'this'

在ES2015中，构造函数的返回值（它是一个对象）隐式地将 `this` 的值替换为 `super()` 的任何调用者。因此，有必要捕获任何潜在的 `super()` 的返回值并替换为 `this` 。此更改允许使用自定义元素，利用此元素可以使用用户编写的构造函数初始化浏览器分配的元素。

示例

```
class Base {
    x: number;
    constructor() {
        // 返回一个除“this”之外的新对象
        return {
            x: 1,
        };
    }
}

class Derived extends Base {
    constructor() {
        super();
        this.x = 2;
    }
}
```

生成：

```
var Derived = (function (_super) {
    __extends(Derived, _super);
    function Derived() {
        var _this = _super.call(this) || this;
        _this.x = 2;
        return _this;
    }
    return Derived;
}(Base));
```

> 这在继承内置类如 `Error` ， `Array` ， `Map` 等的行为上有了破坏性的改变。请阅读extending built-ins breaking change documentation。

配置继承

通常一个项目有多个输出版本，比如 `ES5` 和 `ES2015` ，调试和生产或 `Commonjs` 和 `System` 。只有几个配置选项在这两个版本之间改变，并且维护多个 `tsconfig.json` 文件是麻烦的。

TypeScript 2.1支持使用 `extends` 来继承配置，其中：

- `extends` 在 `tsconfig.json` 是新的顶级属性（与 `compilerOptions` 、 `files` 、 `include` 和 `exclude` 一起）。
- `extends` 的值是包含继承自其它 `tsconfig.json` 路径的字符串。
- 首先加载基本文件中的配置，然后由继承配置文件重写。
- 如果遇到循环，我们报告错误。
- 继承配置文件中的 `files` 、 `include` 和 `exclude` 会重写基本配置文件中相应的值。
- 在配置文件中找到的所有相对路径将相对于它们来源的配置文件来解析。

示例

`configs/base.json` :

```
{
  "compilerOptions": {
    "allowJs": true,
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

`configs/tests.json` :

```
{
  "compilerOptions": {
    "preserveConstEnums": true,
    "stripComments": false,
    "sourceMaps": true
  },
  "exclude": [
    "../tests/baselines",
    "../tests/scenarios"
  ],
  "include": [
    "../tests/**/*.ts"
  ]
}
```

`tsconfig.json` :

```
{
  "extends": "./configs/base",
  "files": [
    "main.ts",
    "supplemental.ts"
  ]
}
```

`tsconfig.nostrictnull.json` :

```
{
  "extends": "./tsconfig",
  "compilerOptions": {
    "strictNullChecks": false
  }
}
```

新编译参数 `--alwaysStrict`

使用 `--alwaysStrict` 调用编译器原因：1.在严格模式下解析的所有代码。2.在每一个生成文件上输出 `"use strict";` 指令;

模块会自动使用严格模式解析。对于非模块代码，建议使用该编译参数。

TypeScript 2.0

Null和undefined类型

TypeScript现在有两个特殊的类型：Null和Undefined, 它们的值分别是 `null` 和 `undefined` 。以前这是不可能明确地命名这些类型的，但是现在 `null` 和 `undefined` 不管在什么类型检查模式下都可以作为类型名称使用。

以前类型检查器认为 `null` 和 `undefined` 赋值给一切。实际上， `null` 和 `undefined` 是每一个类型的有效值， 并且不能明确排除它们（因此不可能检测到错误）。

`--strictNullChecks`

`--strictNullChecks` 可以切换到新的严格空检查模式中。

在严格空检查模式中， `null` 和 `undefined` 值不再属于任何类型的值，仅仅属于它们自己类型和 `any` 类型的值（还有一个例外， `undefined` 也能赋值给 `void` ）。因此，尽管在常规类型检查模式下 `T` 和 `T | undefined` 被认为是相同的（因为 `undefined` 被认为是任何 `T` 的子类型），但是在严格类型检查模式下它们是不同的， 并且仅仅 `T | undefined` 允许有 `undefined` 值， `T` 和 `T | null` 的关系同样如此。

示例

```
// 使用--strictNullChecks参数进行编译的
let x: number;
let y: number | undefined;
let z: number | null | undefined;
x = 1;  // 正确
y = 1;  // 正确
z = 1;  // 正确
x = undefined;  // 错误
y = undefined;  // 正确
z = undefined;  // 正确
x = null;  // 错误
y = null;  // 错误
z = null;  // 正确
x = y;  // 错误
x = z;  // 错误
y = x;  // 正确
y = z;  // 错误
z = x;  // 正确
z = y;  // 正确
```

使用前赋值检查

在严格空检查模式中，编译器要求未包含 `undefined` 类型的局部变量在使用之前必须先赋值。

示例

```
// 使用--strictNullChecks参数进行编译
let x: number;
let y: number | null;
let z: number | undefined;
x;  // 错误，使用前未赋值
y;  // 错误，使用前未赋值
z;  // 正确
x = 1;
y = null;
x;  // 正确
y;  // 正确
```

编译器通过执行基于控制流的类型分析检查变量明确被赋过值。在本篇文章后面会有进一步的细节。

可选参数和属性

可选参数和属性会自动把 `undefined` 添加到他们的类型中，即使他们的类型注解明确不包含 `undefined` 。例如，下面两个类型是完全相同的：

```
// 使用--strictNullChecks参数进行编译
type T1 = (x?: number) => string;              // x的类型是 numbe
r | undefined
type T2 = (x?: number | undefined) => string;  // x的类型是 numbe
r | undefined
```

非null和非undefined类型保护

如果对象或者函数的类型包含 `null` 和 `undefined` ，那么访问属性或调用函数时就会产生编译错误。因此，对类型保护进行了扩展，以支持对非null和非undefined的检查。

示例

```
// 使用--strictNullChecks参数进行编译
declare function f(x: number): string;
let x: number | null | undefined;
if (x) {
    f(x);  // 正确，这里的x类型是number
}
else {
    f(x);  // 错误，这里的x类型是number？
}
let a = x != null ? f(x) : "";  // a的类型是string
let b = x && f(x);  // b的类型是 string | 0 | null | undefined
```

非null和非undefined类型保护可以使用 `==` 、 `!=` 、 `===` 或 `!==` 操作符和 `null` 或 `undefined` 进行比较，如 `x != null` 或 `x === undefined` 。对被试变量类型的影响准确地反映了JavaScript的语义（比如，双等号运算符检查两个值无论你指定的是null还是undefined，然而三等于号运算符仅仅检查指定的那一个值）。

类型保护中的点名称

类型保护以前仅仅支持对局部变量和参数的检查。现在类型保护支持检查由变量或参数名称后跟一个或多个访问属性组成的“点名称”。

示例

```
interface Options {
    location?: {
        x?: number;
        y?: number;
    };
}

function foo(options?: Options) {
    if (options && options.location && options.location.x) {
        const x = options.location.x;  // x的类型是number
    }
}
```

点名称的类型保护和用户定义的类型保护函数，还有 `typeof` 和 `instanceof` 操作符一起工作，并且不依赖 `--strictNullChecks` 编译参数。

对点名称进行类型保护后给点名称任一部分赋值都会导致类型保护无效。例如，对 `x.y.z` 进行了类型保护后给 `x` 、 `x.y` 或 `x.y.z` 赋值，都会导致 `x.y.z` 类型保护无效。

表达式操作符

表达式操作符允许运算对象的类型包含 `null` 和/或 `undefined` ，但是总是产生非null和非undefined类型的结果值。

```
// 使用--strictNullChecks参数进行编译
function sum(a: number | null, b: number | null) {
    return a + b;  // 计算的结果值类型是number
}
```

`&&` 操作符添加 `null` 和/或 `undefined` 到右边操作对象的类型中取决于当前左边操作对象的类型， `||` 操作符从左边联合类型的操作对象的类型中将 `null` 和 `undefined` 同时删除。

```
// 使用--strictNullChecks参数进行编译
interface Entity {
    name: string;
}
let x: Entity | null;
let s = x && x.name;  // s的类型是string | null
let y = x || { name: "test" };  // y的类型是Entity
```

类型扩展

在严格空检查模式中， `null` 和 `undefined` 类型是不会扩展到 `any` 类型中的。

```
let z = null;  // z的类型是null
```

在常规类型检查模式中，由于扩展，会推断 `z` 的类型是 `any` ，但是在严格空检查模式中，推断 `z` 是 `null` 类型（因此，如果没有类型注释， `null` 是 `z` 的唯一值）。

非空断言操作符

在上下文中当类型检查器无法断定类型时，一个新的后缀表达式操作符 `!` 可以用于断言操作对象是非null和非undefined类型的。具体而言，运算 `x!` 产生一个不包含 `null` 和 `undefined` 的 `x` 的值。断言的形式类似于 `<T>x` 和 `x as T` ， `!` 非空断言操作符会从编译成的JavaScript代码中移除。

```
// 使用--strictNullChecks参数进行编译
function validateEntity(e?: Entity) {
    // 如果e是null或者无效的实体，就会抛出异常
}

function processEntity(e?: Entity) {
    validateEntity(e);
    let s = e!.name;  // 断言e是非空并访问name属性
}
```

兼容性

这些新特性是经过设计的，使得它们能够在严格空检查模式和常规类型检查模式下都能够使用。尤其是在常规类型检查模式中， `null` 和 `undefined` 类型会自动从联合类型中删除（因为它们是其它所有类型的子类型）， `!` 非空断言表达式操作符也被允许使用但是没有任何作用。因此，声明文件使用null和undefined敏感类型更新后，在常规类型模式中仍然是可以向后兼容使用的。

在实际应用中，严格空检查模式要求编译的所有文件都是null和undefined敏感类型。

基于控制流的类型分析

TypeScript 2.0实现了对局部变量和参数的控制流类型分析。以前，对类型保护进行类型分析仅限于 `if` 语句和 `?:` 条件表达式，并且不包括赋值和控制流结构的影响，例如 `return` 和 `break` 语句。使用TypeScript 2.0，类型检查器会分析语句和表达式所有可能的控制流，在任何指定的位置对声明为联合类型的局部变量或参数产生最可能的具体类型（缩小范围的类型）。

示例

```
function foo(x: string | number | boolean) {
    if (typeof x === "string") {
        x; // 这里x的类型是string
        x = 1;
        x; // 这里x的类型是number
    }
    x; // 这里x的类型是number | boolean
}

function bar(x: string | number) {
    if (typeof x === "number") {
        return;
    }
    x; // 这里x的类型是string
}
```

基于控制流的类型分析在 `--strictNullChecks` 模式中尤为重要，因为可空类型使用联合类型来表示：

```
function test(x: string | null) {
    if (x === null) {
        return;
    }
    x; // 在函数的剩余部分中，x类型是string
}
```

而且，在 `--strictNullChecks` 模式中，基于控制流的分析包括，对类型不允许为 `undefined` 的局部变量有明确赋值的分析。

```
function mumble(check: boolean) {
    let x: number; // 类型不允许为undefined
    x; // 错误，x是undefined
    if (check) {
        x = 1;
        x; // 正确
    }
    x; // 错误，x可能是undefi
    x = 2;
    x; // 正确
}
```

标记联合类型

TypeScript 2.0实现了标记（或区分）联合类型。具体而言，TS编译器现在支持类型保护，基于判别属性的检查来缩小联合类型的范围，并且 `switch` 语句也支持此特性。

示例

```
interface Square {
    kind: "square";
    size: number;
}

interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}

interface Circle {
    kind: "circle";
    radius: number;
}

type Shape = Square | Rectangle | Circle;
```

```
function area(s: Shape) {
    // 在下面的switch语句中，s的类型在每一个case中都被缩小
    // 根据判别属性的值，变量的其它属性不使用类型断言就可以被访问
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.width * s.height;
        case "circle": return Math.PI * s.radius * s.radius;
    }
}

function test1(s: Shape) {
    if (s.kind === "square") {
        s;  // Square
    }
    else {
        s;  // Rectangle | Circle
    }
}

function test2(s: Shape) {
    if (s.kind === "square" || s.kind === "rectangle") {
        return;
    }
    s;  // Circle
}
```

判别属性类型保护是 `x.p == v` 、 `x.p === v` 、 `x.p != v` 或者 `x.p !== v` 其中的一种表达式， `p` 和 `v` 是一个属性和字符串字面量类型或字符串字面量联合类型的表达式。判别属性类型保护缩小 `x` 的类型到由判别属性 `p` 和 `v` 的可能值之一组成的类型。

请注意，我们目前只支持字符串字面值类型的判别属性。我们打算以后添加对布尔值和数字字面量类型的支持。

`never` 类型

TypeScript 2.0引入了一个新原始类型 `never` 。 `never` 类型表示值的类型从不出现。具体而言， `never` 是永不返回函数的返回类型，也是变量在类型保护中永不为true的类型。

`never` 类型具有以下特征：

- `never` 是所有类型的子类型并且可以赋值给所有类型。
- 没有类型是 `never` 的子类型或能赋值给 `never` （ `never` 类型本身除外）。
- 在函数表达式或箭头函数没有返回类型注解时，如果函数没有 `return` 语句，或者只有 `never` 类型表达式的 `return` 语句，并且如果函数是不可执行到终点的（例如通过控制流分析决定的），则推断函数的返回类型是 `never` 。
- 在有明确 `never` 返回类型注解的函数中，所有 `return` 语句（如果有的话）必须有 `never` 类型的表达式并且函数的终点必须是不可执行的。

因为 `never` 是每一个类型的子类型，所以它总是在联合类型中被省略，并且在函数中只要其它类型被返回，类型推断就会忽略 `never` 类型。

一些返回 `never` 函数的示例：

```
// 函数返回never必须无法执行到终点
function error(message: string): never {
    throw new Error(message);
}

// 推断返回类型是never
function fail() {
    return error("Something failed");
}

// 函数返回never必须无法执行到终点
function infiniteLoop(): never {
    while (true) {
    }
}
```

一些函数返回 `never` 的使用示例：

```
// 推断返回类型是number
function move1(direction: "up" | "down") {
    switch (direction) {
        case "up":
            return 1;
        case "down":
            return -1;
    }
    return error("Should never get here");
}

// 推断返回类型是number
function move2(direction: "up" | "down") {
    return direction === "up" ? 1 :
        direction === "down" ? -1 :
        error("Should never get here");
}

// 推断返回类型是T
function check<T>(x: T | undefined) {
    return x || error("Undefined value");
}
```

因为 `never` 可以赋值给每一个类型，当需要回调函数返回一个更加具体的类型时，函数返回 `never` 类型可以用于检测返回类型是否正确：

```
function test(cb: () => string) {
    let s = cb();
    return s;
}

test(() => "hello");
test(() => fail());
test(() => { throw new Error(); })
```

只读属性和索引签名

属性或索引签名现在可以使用 `readonly` 修饰符声明为只读的。

只读属性可以初始化和在同一个类的构造函数中被赋值，但是在其它情况下对只读属性的赋值是不允许的。

此外，有几种情况下实体隐式只读的：

- 属性声明只使用 `get` 访问器而没有使用 `set` 访问器被视为只读的。
- 在枚举类型中，枚举成员被视为只读属性。
- 在模块类型中，导出的 `const` 变量被视为只读属性。
- 在 `import` 语句中声明的实体被视为只读的。
- 通过ES2015命名空间导入访问的实体被视为只读的（例如，当 `foo` 当作 `import * as foo from "foo"` 声明时，`foo.x` 是只读的）。

示例

```
interface Point {
    readonly x: number;
    readonly y: number;
}

var p1: Point = { x: 10, y: 20 };
p1.x = 5;  // 错误，p1.x是只读的

var p2 = { x: 1, y: 1 };
var p3: Point = p2;  // 正确，p2的只读别名
p3.x = 5;  // 错误，p3.x是只读的
p2.x = 5;  // 正确，但是因为别名使用，同时也改变了p3.x
```

```
class Foo {
    readonly a = 1;
    readonly b: string;
    constructor() {
        this.b = "hello";  // 在构造函数中允许赋值
    }
}
```

```
let a: Array<number> = [0, 1, 2, 3, 4];
let b: ReadonlyArray<number> = a;
b[5] = 5;       // 错误，元素是只读的
b.push(5);      // 错误，没有push方法（因为这会修改数组）
b.length = 3;   // 错误，length是只读的
a = b;          // 错误，缺少修改数组的方法
```

指定函数中 `this` 类型

紧跟着类和接口，现在函数和方法也可以声明 `this` 的类型了。

函数中 `this` 的默认类型是 `any` 。从TypeScript 2.0开始，你可以提供一个明确的 `this` 参数。 `this` 参数是伪参数，它位于函数参数列表的第一位：

```
function f(this: void) {
    // 确保`this`在这个独立的函数中无法使用
}
```

回调函数中的 `this` 参数

库也可以使用 `this` 参数声明回调函数如何被调用。

示例

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): v
oid;
}
```

`this:void` 意味着 `addClickListener` 预计 `onclick` 是一个 `this` 参数不需要类型的函数。

现在如果你在调用代码中对 `this` 进行了类型注释：

```
class Handler {
    info: string;
    onClickBad(this: Handler, e: Event) {
        // 哎哟，在这里使用this.在运行中使用这个回调函数将会崩溃。
        this.info = e.message;
    };
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // 错误！
```

`--noImplicitThis`

TypeScript 2.0还增加了一个新的编译选项用来标记函数中所有没有明确类型注释的 `this` 的使用。

`tsconfig.json` 支持文件通配符

文件通配符来啦！！支持文件通配符一直是最需要的特性之一。

类似文件通配符的文件模式支持两个属性 `"include"` 和 `"exclude"` 。

示例

```
{
    "compilerOptions": {
        "module": "commonjs",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "outFile": "../../built/local/tsc.js",
        "sourceMap": true
    },
    "include": [
        "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}
```

支持文件通配符的符号有：

- `*` 匹配零个或多个字符（不包括目录）
- `?` 匹配任意一个字符（不包括目录）
- `**/` 递归匹配所有子目录

如果文件通配符模式语句中只包含 `*` 或 `.*` ，那么只匹配带有扩展名的文件（例如默认是 `.ts` 、 `.tsx` 和 `.d.ts` ，如果 `allowJs` 设置为 `true` ， `.js` 和 `.jsx` 也属于默认）。

如果 `"files"` 和 `"include"` 都没有指定，编译器默认包含所有目录中的TypeScript文件（ `.ts` 、 `.d.ts` 和 `.tsx` ），除了那些使用 `exclude` 属性排除的文件外。如果 `allowJs` 设置为true，JS文件（ `.js` 和 `.jsx` ）也会被包含进去。

如果 `"files"` 和 `"include"` 都指定了，编译器将包含这两个属性指定文件的并集。使用 `ourDir` 编译选项指定的目录文件总是被排除，即使 `"exclude"` 属性指定的文件也会被删除，但是 `files` 属性指定的文件不会排除。

`"exclude"` 属性指定的文件会对 `"include"` 属性指定的文件过滤。但是对 `"files"` 指定的文件没有任何作用。当没有明确指定时， `"exclude"` 属性默认会排除 `node_modules` 、 `bower_components` 和 `jspm_packages` 目录。

模块解析增加：BaseUrl、路径映射、rootDirs和追踪

TypeScript 2.0提供了一系列额外的模块解析属性告诉编译器去哪里可以找到给定模块的声明。

更多详情，请参阅模块解析文档。

Base URL

使用了AMD模块加载器并且模块在运行时"部署"到单文件夹的应用程序中使用 `baseUrl` 是一种常用的做法。所有非相对名称的模块导入被认为是相对于 `baseUrl` 的。

示例

```
{
  "compilerOptions": {
    "baseUrl": "./modules"
  }
}
```

现在导入 `moduleA` 将会在 `./modules/moduleA` 中查找。

```
import A from "moduleA";
```

路径映射

有时模块没有直接位于*baseUrl*中。加载器使用映射配置在运行时去映射模块名称和文件，请参阅RequireJs文档和SystemJS文档。

TypeScript编译器支持 `tsconfig` 文件中使用 `"paths"` 属性映射的声明。

示例

例如，导入 `"jquery"` 模块在运行时会被转换为 `"node_modules/jquery/dist/jquery.slim.min.js"` 。

```
{
    "compilerOptions": {
        "baseUrl": "./node_modules",
        "paths": {
        "jquery": ["jquery/dist/jquery.slim.min"]
        }
    }
}
```

使用 `"paths"` 也允许更复杂的映射，包括多次后退的位置。考虑一个只有一个地方的模块是可用的，其它的模块都在另一个地方的项目配置。

`rootDirs` 和虚拟目录

使用 `rootDirs` ，你可以告知编译器的根目录组合这些“虚拟”目录。因此编译器在这些“虚拟”目录中解析相对导入模块，仿佛是合并到一个目录中一样。

示例

给定的项目结构

```
src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
        └── views
            └── template1.ts (imports './view2')
```

构建步骤将复制 `/src/views` 和 `/generated/templates/views` 目录下的文件输出到同一个目录中。在运行时，视图期望它的模板和它存在同一目录中，因此应该使用相对名称 `"./template"` 导入。

`"rootDir"` 指定的一组根目录的内容将会在运行时合并。因此在我们的例子， `tsconfig.json` 文件应该类似于：

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

追踪模块解析

`--traceResolution` 提供了一种方便的方法，以了解模块如何被编译器解析的。

```
tsc --traceResolution
```

快捷外部模块声明

当你使用一个新模块时，如果不想要花费时间书写一个声明时，现在你可以使用快捷声明以便以快速开始。

declarations.d.ts

```
declare module "hot-new-module";
```

所有从快捷模块的导入都具有任意类型。

```
import x, {y} from "hot-new-module";
x(y);
```

模块名称中的通配符

以前使用模块加载器（例如AMD和SystemJS）导入没有代码的资源是不容易的。之前，必须为每个资源定义一个外部模块声明。

TypeScript 2.0支持使用通配符符号（ `*` ）定义一类模块名称。这种方式，一个声明只需要一次扩展名，而不再是每一个资源。

示例

```
declare module "*!text" {
    const content: string;
    export default content;
}
// Some do it the other way around.
declare module "json!*" {
    const value: any;
    export default value;
}
```

现在你可以导入匹配 `"*!text"` 或 `"json!*"` 的东西了。

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

当从一个基于非类型化的代码迁移时，通配符模块的名称可能更加有用。结合快捷外部模块声明，一组模块可以很容易地声明为 `any` 。

示例

```
declare module "myLibrary/*";
```

所有位于 `myLibrary` 目录之下的模块的导入都被编译器认为是 `any` 类型，因此这些模块的任何类型检查都会被关闭。

```
import { readFile } from "myLibrary/fileSystem/readFile`;

readFile(); // readFile是'any'类型
```

支持UMD模块定义

一些库被设计为可以使用多种模块加载器或者不是使用模块加载器（全局变量）来使用，这被称为UMD或同构模块。这些库可以通过导入或全局变量访问。

举例：

math-lib.d.ts

```
export const isPrime(x: number): boolean;
export as namespace mathLib;
```

然后，该库可作为模块导入使用：

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // 错误：无法在模块内部使用全局定义
```

它也可以被用来作为一个全局变量，只限于没有 `import` 和 `export` 脚本文件中。

```
mathLib.isPrime(2);
```

可选类属性

现在可以在类中声明可选属性和方法，与接口类似。

示例

```
class Bar {
    a: number;
    b?: number;
    f() {
        return 1;
    }
    g?(): number;  // 可选方法的方法体可以省略
    h?() {
        return 2;
    }
}
```

在 `--strictNullChecks` 模式下编译时，可选属性和方法会自动添加 `undefined` 到它们的类型中。因此，上面的 `b` 属性类型是 `number | undefined`，上面 `g` 方法的类型是 `(()=> number) | undefined`。使用类型保护可以去除 `undefined`。

私有的和受保护的构造函数

类的构造函数可以被标记为 `private` 或 `protected`。私有构造函数的类不能在类的外部实例化，并且也不能被继承。受保护构造函数的类不能再类的外部实例化，但是可以被继承。

示例

```
class Singleton {
    private static instance: Singleton;

    private constructor() { }

    static getInstance() {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}

let e = new Singleton(); // 错误：Singleton的构造函数是私有的。
let v = Singleton.getInstance();
```

抽象属性和访问器

抽象类可以声明抽象属性和、或访问器。所有子类将需要声明抽象属性或者被标记为抽象的。抽象属性不能初始化。抽象访问器不能有具体代码块。

示例

```
abstract class Base {
    abstract name: string;
    abstract get value();
    abstract set value(v: number);
}

class Derived extends Base {
    name = "derived";

    value = 1;
}
```

隐式索引签名

如果对象字面量中所有已知的属性是赋值给索引签名，那么现在对象字面量类型可以赋值给索引签名类型。这使得一个使用对象字面量初始化的变量作为参数传递给期望参数是map或dictionary的函数成为可能：

```
function httpService(path: string, headers: { [x: string]: string
 }) { }

const headers = {
    "Content-Type": "application/x-www-form-urlencoded"
};

httpService("", { "Content-Type": "application/x-www-form-urlencoded" });  // 可以
httpService("", headers);  // 现在可以，以前不可以。
```

使用 `--lib` 编译参数包含内置类型声明

获取ES6/ES2015内置API声明仅限于 `target: ES6` 。输入 `--lib` ，你可以使用 `--lib` 指定一组项目所需要的内置API。比如说，如果你希望项目运行时支持 `Map` 、 `Set` 和 `Promise` （例如现在静默更新浏览器），直接写 `--lib es2015.collection,es2015.promise` 就好了。同样，你也可以排除项目中不需要的声明，例如在node项目中使用 `--lib es5,es6` 排除DOM。

下面是列出了可用的API：

- dom
- webworker
- es5
- es6 / es2015
- es2015.core
- es2015.collection
- es2015.iterable
- es2015.promise
- es2015.proxy
- es2015.reflect
- es2015.generator

- es2015.symbol
- es2015.symbol.wellknown
- es2016
- es2016.array.include
- es2017
- es2017.object
- es2017.sharedmemory
- scripthost

示例

```
tsc --target es5 --lib es5,es2015.promise
```

```
"compilerOptions": {
    "lib": ["es5", "es2015.promise"]
}
```

使用 `--noUnusedParameters` 和 `--noUnusedLocals` 标记未使用的声明

TypeScript 2.0有两个新的编译参数来帮助你保持一个干净的代码库。 `-noUnusedParameters` 编译参数标记所有未使用的函数或方法的参数错误。 `--noUnusedLocals` 标记所有未使用的局部（未导出）声明像变量、函数、类和导入等等，另外未使用的私有类成员在 `--noUnusedLocals` 作用下也会标记为错误。

示例

```
import B, { readFile } from "./b";
//     ^ 错误：`B`声明了，但是没有使用。
readFile();


export function write(message: string, args: string[]) {
    //                                 ^^^^  错误：'arg'声明了，但是没有使用。
    console.log(message);
}
```

使用以 `_` 开头命名的参数声明不会被未使用参数检查。例如：

```
function returnNull(_a) { // 正确
    return null;
}
```

模块名称允许 `.js` 扩展名

TypeScript 2.0之前，模块名称总是被认为是没有扩展名的。例如，导入一个模块 `import d from "./moduleA.js"` ，则编译器在 `./moduleA.js.ts` 或 `./moduleA.js.d.ts` 中查找 `"moduleA.js"` 的定义。这使得像SystemJS这种期望模块名称是URI的打包或加载工具很难使用。

使用TypeScript 2.0，编译器将在 `./moduleA.ts` 或 `./moduleA.d.ts` 中查找 `"moduleA.js"` 的定义。

支持编译参数 `target : es5` 和 `module: es6` 同时使用

之前编译参数 `target : es5` 和 `module: es6` 同时使用被认为是无效的，但是现在是有效的。这将有助于使用基于ES2015的tree-shaking（将无用代码移除）比如rollup。

函数形参和实参列表末尾支持逗号

现在函数形参和实参列表末尾允许有逗号。这是对第三阶段的ECMAScript提案的实现，并且会编译为可用的 ES3/ES5/ES6。

示例

```
function foo(
  bar: Bar,
  baz: Baz, // 形参列表末尾添加逗号是没有问题的。
) {
  // 具体实现……
}

foo(
  bar,
  baz, // 实参列表末尾添加逗号同样没有问题
);
```

新编译参数 `--skipLibCheck`

TypeScript 2.0添加了一个新的编译参数 `--skipLibCheck` ，该参数可以跳过声明文件（以 `.d.ts` 为扩展名的文件）的类型检查。当一个程序包含有大量的声明文件时，编译器需要花费大量时间对已知不包含错误的声明进行类型检查，通过跳过声明文件的类型检查，编译时间可能会大大缩短。

由于一个文件中的声明可以影响其他文件中的类型检查，当指定 `--skipLibCheck` 时，一些错误可能检测不到。比如说，如果一个非声明文件中的类型被声明文件用到，可能仅在声明文件被检查时能发现错误.不过这种情况在实际使用中并不常见。

允许在声明中重复标识符

这是重复定义错误的一个常见来源。多个声明文件定义相同的接口成员。

TypeScript 2.0放宽了这一约束，并允许可以不同代码块中出现重复的标识符，只要它们有完全相同的类型。

在同一代码块重复定义仍不允许。

示例

```
interface Error {
    stack?: string;
}


interface Error {
    code?: string;
    path?: string;
    stack?: string;  // OK
}
```

新编译参数`--declarationDir`

`--declarationDir`可以使生成的声明文件和JavaScript文件不在同一个位置中。

TypeScript 1.8

类型参数约束

在 TypeScript 1.8 中, 类型参数的限制可以引用自同一个类型参数列表中的类型参数. 在此之前这种做法会报错. 这种特性通常被叫做 F-Bounded Polymorphism.

例子

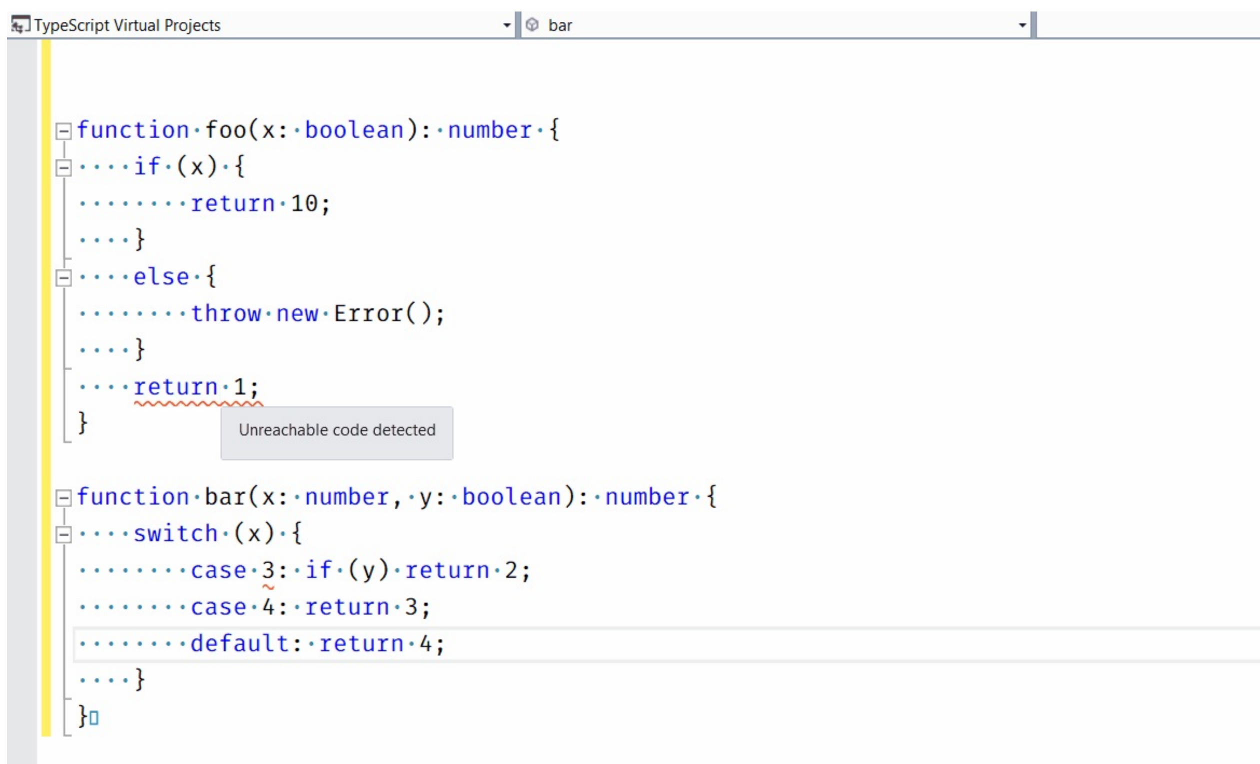
```
function assign<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = source[id];
    }
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 });  // 错误
```

控制流错误分析

TypeScript 1.8 中引入了控制流分析来捕获开发者通常会遇到的一些错误.

详情见接下来的内容, 可以上手尝试:



不可及的代码

一定无法在运行时被执行的语句现在会被标记上代码不可及错误. 举个例子, 在无条件限制的 `return`, `throw`, `break` 或者 `continue` 后的语句被认为是不可及的. 使用 `--allowUnreachableCode` 来禁用不可及代码的检测和报错.

例子

这里是一个简单的不可及错误的例子:

```
function f(x) {
    if (x) {
       return true;
    }
    else {
       return false;
    }

    x = 0; // 错误: 检测到不可及的代码.
}
```

这个特性能捕获的一个更常见的错误是在 `return` 语句后添加换行:

```
function f() {
    return            // 换行导致自动插入的分号
    {
        x: "string"   // 错误: 检测到不可及的代码.
    }
}
```

因为 JavaScript 会自动在行末结束 `return` 语句, 下面的对象字面量变成了一个代码块.

未使用的标签

未使用的标签也会被标记. 和不可及代码检查一样, 被使用的标签检查也是默认开启的. 使用 `--allowUnusedLabels` 来禁用未使用标签的报错.

例子

```
loop: while (x > 0) {  // 错误: 未使用的标签.
    x++;
}
```

隐式返回

JS 中没有返回值的代码分支会隐式地返回 `undefined` . 现在编译器可以将这种方式标记为隐式返回. 对于隐式返回的检查默认是被禁用的, 可以使用 `--noImplicitReturns` 来启用.

例子

```
function f(x) { // 错误: 不是所有分支都返回了值.
    if (x) {
        return false;
    }

    // 隐式返回了 `undefined`
}
```

Case 语句贯穿

TypeScript 现在可以在 switch 语句中出现贯穿的几个非空 case 时报错. 这个检测默认是关闭的, 可以使用 `--noFallthroughCasesInSwitch` 启用.

例子

```
switch (x % 2) {
    case 0: // 错误: switch 中出现了贯穿的 case.
        console.log("even");

    case 1:
        console.log("odd");
        break;
}
```

然而, 在下面的例子中, 由于贯穿的 case 是空的, 并不会报错:

```
switch (x % 3) {
    case 0:
    case 1:
        console.log("Acceptable");
        break;

    case 2:
        console.log("This is *two much*!");
        break;
}
```

React 无状态的函数组件

TypeScript 现在支持无状态的函数组件. 它是可以组合其他组件的轻量级组件.

```
// 使用参数解构和默认值轻松地定义 'props' 的类型
const Greeter = ({name = 'world'}) => <div>Hello, {name}!</div>;

// 参数可以被检验
let example = <Greeter name='TypeScript 1.8' />;
```

如果需要使用这一特性及简化的 props, 请确认使用的是最新的 react.d.ts.

简化的 React `props` 类型管理

在 TypeScript 1.8 配合最新的 react.d.ts (见上方) 大幅简化了 `props` 的类型声明.

具体的:

- 你不再需要显式的声明 `ref` 和 `key` 或者 `extend React.Props`
- `ref` 和 `key` 属性会在所有组件上拥有正确的类型.
- `ref` 属性在无状态函数组件上会被正确地禁用.

在模块中扩充全局或者模块作用域

用户现在可以为任何模块进行他们想要, 或者其他人已经对其作出的扩充. 模块扩充的形式和过去的包模块一致 (例如 `declare module "foo" { }` 这样的语法), 并且可以直接嵌在你自己的模块内, 或者在另外的顶级外部包模块中.

除此之外, TypeScript 还以 `declare global { }` 的形式提供了对于全局声明的扩充. 这能使模块对像 `Array` 这样的全局类型在必要的时候进行扩充.

模块扩充的名称解析规则与 `import` 和 `export` 声明中的一致. 扩充的模块声明合并方式与在同一个文件中声明是相同的.

不论是模块扩充还是全局声明扩充都不能向顶级作用域添加新的项目 - 它们只能为已经存在的声明添加 "补丁".

例子

这里的 `map.ts` 可以声明它会在内部修改在 `observable.ts` 中声明的 `Observable` 类型, 添加 `map` 方法.

```
// observable.ts
export class Observable<T> {
    // ...
}
```

```
// map.ts
import { Observable } from "./observable";

// 扩充 "./observable"
declare module "./observable" {

    // 使用接口合并扩充 'Observable' 类的定义
    interface Observable<T> {
        map<U>(proj: (el: T) => U): Observable<U>;
    }

}

Observable.prototype.map = /*...*/;
```

```
// consumer.ts
import { Observable } from "./observable";
import "./map";

let o: Observable<number>;
o.map(x => x.toFixed());
```

相似的, 在模块中全局作用域可以使用 `declare global` 声明被增强:

例子

```
// 确保当前文件被当做一个模块.
export {};

declare global {
    interface Array<T> {
        mapToNumbers(): number[];
    }
}

Array.prototype.mapToNumbers = function () { /* ... */ }
```

字符串字面量类型

接受一个特定字符串集合作为某个值的 API 并不少见. 举例来说, 考虑一个可以通过控制动画的渐变让元素在屏幕中滑动的 UI 库:

```
declare class UIElement {
    animate(options: AnimationOptions): void;
}

interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: string; // 可以是 "ease-in", "ease-out", "ease-in-out"

}
```

然而, 这容易产生错误 - 当用户错误不小心错误拼写了一个合法的值时, 并没有任何提示:

```
// 没有报错
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "eas
e-inout" });
```

在 TypeScript 1.8 中, 我们新增了字符串字面量类型. 这些类型和字符串字面量的写法一致, 只是写在类型的位置.

用户现在可以确保类型系统会捕获这样的错误. 这里是我们使用了字符串字面量类型的新的 `AnimationOptions` :

```
interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: "ease-in" | "ease-out" | "ease-in-out";
}

// 错误: 类型 '"ease-inout"' 不能复制给类型 '"ease-in" | "ease-out" | "ease-in-out"'
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-inout" });
```

更好的联合/交叉类型接口

TypeScript 1.8 优化了源类型和目标类型都是联合或者交叉类型的情况下的类型推导. 举例来说, 当从 `string | string[]` 推导到 `string | T` 时, 我们将类型拆解为 `string[]` 和 `T` , 这样就可以将 `string[]` 推导为 `T` .

例子

```
type Maybe<T> = T | void;

function isDefined<T>(x: Maybe<T>): x is T {
    return x !== undefined && x !== null;
}

function isUndefined<T>(x: Maybe<T>): x is void {
    return x === undefined || x === null;
}

function getOrElse<T>(x: Maybe<T>, defaultValue: T): T {
    return isDefined(x) ? x : defaultValue;
}

function test1(x: Maybe<string>) {
    let x1 = getOrElse(x, "Undefined");         // string
    let x2 = isDefined(x) ? x : "Undefined";    // string
    let x3 = isUndefined(x) ? "Undefined" : x;  // string
}

function test2(x: Maybe<number>) {
    let x1 = getOrElse(x, -1);         // number
    let x2 = isDefined(x) ? x : -1;    // number
    let x3 = isUndefined(x) ? -1 : x;  // number
}
```

使用 `--outFile` 合并 `AMD` 和 `System` 模块

在使用 `--module amd` 或者 `--module system` 的同时制定 `--outFile` 将会把所有参与编译的模块合并为单个包括了多个模块闭包的输出文件.

每一个模块都会根据其相对于 `rootDir` 的位置被计算出自己的模块名称.

例子

```
// 文件 src/a.ts
import * as B from "./lib/b";
export function createA() {
    return B.createB();
}
```

```
// 文件 src/lib/b.ts
export function createB() {
    return { };
}
```

结果为:

```
define("lib/b", ["require", "exports"], function (require, expor
ts) {
    "use strict";
    function createB() {
        return {};
    }
    exports.createB = createB;
});
define("a", ["require", "exports", "lib/b"], function (require,
exports, B) {
    "use strict";
    function createA() {
        return B.createB();
    }
    exports.createA = createA;
});
```

支持 SystemJS 使用 `default` 导入

像 SystemJS 这样的模块加载器将 CommonJS 模块做了包装并暴露为 `default` ES6 导入项. 这使得在 SystemJS 和 CommonJS 的实现由于不同加载器不同的模块导出方式不能共享定义.

设置新的编译选项 `--allowSyntheticDefaultImports` 指明模块加载器会进行导入的 `.ts` 或 `.d.ts` 中未指定的某种类型的默认导入项构建. 编译器会由此推断存在一个 `default` 导出项和整个模块自己一致.

此选项在 System 模块默认开启.

允许循环中被引用的 `let` / `const`

之前这样会报错, 现在由 TypeScript 1.8 支持. 循环中被函数引用的 `let` / `const` 声明现在会被输出为与 `let` / `const` 更新语义相符的代码.

例子

```
let list = [];
for (let i = 0; i < 5; i++) {
    list.push(() => i);
}

list.forEach(f => console.log(f()));
```

被编译为:

```
var list = [];
var _loop_1 = function(i) {
    list.push(function () { return i; });
};
for (var i = 0; i < 5; i++) {
    _loop_1(i);
}
list.forEach(function (f) { return console.log(f()); });
```

然后结果是:

```
0
1
2
3
4
```

改进的 `for..in` 语句检查

过去 `for..in` 变量的类型被推断为 `any`，这使得编译器忽略了 `for..in` 语句内的一些不合法的使用.

从 TypeScript 1.8 开始:

- 在 `for..in` 语句中的变量隐含类型为 `string`.
- 当一个有数字索引签名对应类型 `T` (比如一个数组) 的对象被一个 `for..in` 索引有数字索引签名并且没有字符串索引签名 (比如还是数组) 的对象的变量索引，产生的值的类型为 `T`.

例子

```
var a: MyObject[];
for (var x in a) {   // x 的隐含类型为 string
    var obj = a[x];  // obj 的类型为 MyObject
}
```

模块现在输出时会加上 `"use strict;"`

对于 ES6 来说模块始终以严格模式被解析，但这一点过去对于非 ES6 目标在生成的代码中并没有遵循. 从 TypeScript 1.8 开始，输出的模块总会为严格模式. 由于多数严格模式下的错误也是 TS 编译时的错误，多数代码并不会有可见的改动，但是这也意味着有一些东西可能在运行时没有征兆地失败，比如赋值给 `NaN` 现在会有运行时错误. 你可以参考这篇 MDN 上的文章 查看详细的严格模式与非严格模式的区别列表.

使用 `--allowJs` 加入 `.js` 文件

经常在项目中会有外部的非 TypeScript 编写的源文件. 一种方式是将 JS 代码转换为 TS 代码, 但这时又希望将所有 JS 代码和新的 TS 代码的输出一起打包为一个文件.

`.js` 文件现在允许作为 `tsc` 的输入文件. TypeScript 编译器会检查 `.js` 输入文件的语法错误, 并根据 `--target` 和 `--module` 选项输出对应的代码. 输出也会和其他 `.ts` 文件一起. `.js` 文件的 source maps 也会像 `.ts` 文件一样被生成.

使用 `--reactNamespace` 自定义 JSX 工厂

在使用 `--jsx react` 的同时使用 `--reactNamespace <JSX 工厂名称>` 可以允许使用一个不同的 JSX 工厂代替默认的 `React` .

新的工厂名称会被用来调用 `createElement` 和 `__spread` 方法.

例子

```
import {jsxFactory} from "jsxFactory";

var div = <div>Hello JSX!</div>
```

编译参数:

```
tsc --jsx react --reactNamespace jsxFactory --m commonJS
```

结果:

```
"use strict";
var jsxFactory_1 = require("jsxFactory");
var div = jsxFactory_1.jsxFactory.createElement("div", null, "Hello JSX!");
```

基于 `this` 的类型收窄

TypeScript 1.8 为类和接口方法扩展了用户定义的类型收窄函数.

`this is T` 现在是类或接口方法的合法的返回值类型标注. 当在类型收窄的位置使用时 (比如 `if` 语句), 函数调用表达式的目标对象的类型会被收窄为 `T`.

例子

```
class FileSystemObject {
    isFile(): this is File { return this instanceof File; }
    isDirectory(): this is Directory { return this instanceof Di
rectory;}
    isNetworked(): this is (Networked & this) { return this.netw
orked; }
    constructor(public path: string, private networked: boolean)
 {}
}

class File extends FileSystemObject {
    constructor(path: string, public content: string) { super(pa
th, false); }
}
class Directory extends FileSystemObject {
    children: FileSystemObject[];
}
interface Networked {
    host: string;
}

let fso: FileSystemObject = new File("foo/bar.txt", "foo");
if (fso.isFile()) {
    fso.content; // fso 是 File
}
else if (fso.isDirectory()) {
    fso.children; // fso 是 Directory
}
else if (fso.isNetworked()) {
    fso.host; // fso 是 networked
}
```

官方的 TypeScript NuGet 包

从 TypeScript 1.8 开始, 将为 TypeScript 编译器 (`tsc.exe`) 和 MSBuild 整合 (`Microsoft.TypeScript.targets` 和 `Microsoft.TypeScript.Tasks.dll`) 提供官方的 NuGet 包.

稳定版本可以在这里下载:

- Microsoft.TypeScript.Compiler
- Microsoft.TypeScript.MSBuild

与此同时,和每日npm包对应的每日 NuGet 包可以在https://myget.org下载:

- TypeScript-Preview

`tsc` 错误信息更美观

我们理解大量单色的输出并不直观. 颜色可以帮助识别信息的始末, 这些视觉上的线索在处理复杂的错误信息时非常重要.

通过传递 `--pretty` 命令行选项, TypeScript 会给出更丰富的输出, 包含错误发生的上下文.

```
Windows PowerShell (Admin)
<1> Windows PowerSh...
PS C:\iFeelPretty> tsc --watch --pretty

18     await reticulateSplines();
       ~~~~~

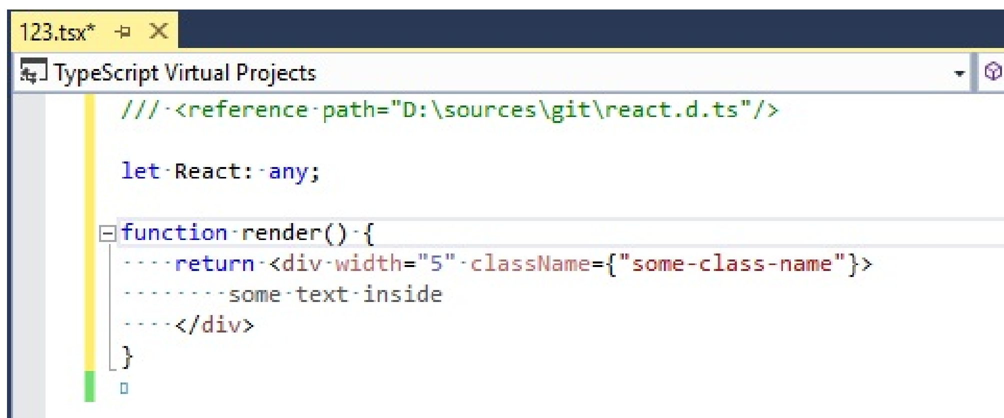
ohSoPretty.ts(18,5): error TS1308: 'await' expression is only allowed within an async function.

13:32:44 - Compilation complete. Watching for file changes.

node.exe[*32]:4892        « 150513[64]  1/1  [+] NUM  PRI↕  99x23  (1,9) 25V  15108 100%
```

高亮 VS 2015 中的 JSX 代码

在 TypeScript 1.8 中, JSX 标签现在可以在 Visual Studio 2015 中被分别和高亮.



通过 `工具` -> `选项` -> `环境` -> `字体与颜色` 页面在 `VB XML` 颜色和字体设置中还可以进一步改变字体和颜色来自定义.

`--project` (`-p`) 选项现在接受任意文件路径

`--project` 命令行选项过去只接受包含了 `tsconfig.json` 文件的文件夹. 考虑到不同的构建场景, 应该允许 `--project` 指向任何兼容的 JSON 文件. 比如说, 一个用户可能会希望为 Node 5 编译 CommonJS 的 ES 2015, 为浏览器编译 AMD 的 ES5. 现在少了这项限制, 用户可以更容易地直接使用 `tsc` 管理不同的构建目标, 无需再通过一些奇怪的方式, 比如将多个 `tsconfig.json` 文件放在不同的目录中.

如果参数是一个路径, 行为保持不变 - 编译器会尝试在该目录下寻找名为 `tsconfig.json` 的文件.

允许 tsconfig.json 中的注释

为配置添加文档是很棒的! `tsconfig.json` 现在支持单行和多行注释.

```
{
    "compilerOptions": {
        "target": "ES2015", // 跑在 node v5 上, 呀!
        "sourceMap": true   // 让调试轻松一些
    },
    /*
     * 排除的文件
     */
    "exclude": [
        "file.d.ts"
    ]
}
```

支持输出到 IPC 驱动的文件

TypeScript 1.8 允许用户将 `--outFile` 参数和一些特殊的文件系统对象一起使用, 比如命名的管道 (pipe), 设备 (devices) 等.

举个例子, 在很多与 Unix 相似的系统上, 标准输出流可以通过文件 `/dev/stdout` 访问.

```
tsc foo.ts --outFile /dev/stdout
```

这一特性也允许输出给其他命令.

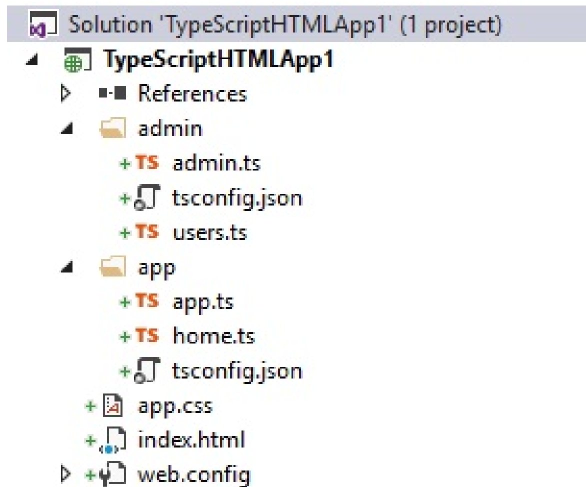
比如说, 我们可以输出生成的 JavaScript 给一个像 pretty-js 这样的格式美化工具:

```
tsc foo.ts --outFile /dev/stdout | pretty-js
```

改进了 Visual Studio 2015 中对 `tsconfig.json` 的支持

TypeScript 1.8 允许在任何种类的项目中使用 `tsconfig.json` 文件. 包括 ASP.NET v4 项目, 控制台应用, 以及 用 *TypeScript* 开发的 *HTML* 应用. 与此同时, 你可以添加不止一个 `tsconfig.json` 文件, 其中每一个都会作为项目的一部分被

构建. 这使得你可以在不使用多个不同项目的情况下为应用的不同部分使用不同的配置.



当项目中添加了 `tsconfig.json` 文件时, 我们还禁用了项目属性页面. 也就是说所有配置的改变必须在 `tsconfig.json` 文件中进行.

一些限制

- 如果你添加了一个 `tsconfig.json` 文件, 不在其上下文中的 TypeScript 文件不会被编译.
- Apache Cordova 应用依然有单个 `tsconfig.json` 文件的限制, 而这个文件必须在根目录或者 `scripts` 文件夹.
- 多数项目类型中都没有 `tsconfig.json` 的模板.

TypeScript 1.7

支持 `async` / `await` 编译到 ES6 (Node v4+)

TypeScript 目前在已经原生支持 ES6 generator 的引擎 (比如 Node v4 及以上版本) 上支持异步函数. 异步函数前置 `async` 关键字; `await` 会暂停执行, 直到一个异步函数执行后返回的 promise 被 fulfill 后获得它的值.

例子

在下面的例子中, 输入的内容将会延时 200 毫秒逐个打印:

```
"use strict";

// printDelayed 返回值是一个 'Promise<void>'
async function printDelayed(elements: string[]) {
    for (const element of elements) {
        await delay(200);
        console.log(element);
    }
}

async function delay(milliseconds: number) {
    return new Promise<void>(resolve => {
        setTimeout(resolve, milliseconds);
    });
}

printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {
    console.log();
    console.log("打印每一个内容!");
});
```

查看 Async Functions 一文了解更多.

支持同时使用 `--target ES6` 和 `--module`

TypeScript 1.7 将 `ES6` 添加到了 `--module` 选项支持的选项的列表, 当编译到 `ES6` 时允许指定模块类型. 这让使用具体运行时中你需要的特性更加灵活.

例子

```
{
    "compilerOptions": {
        "module": "amd",
        "target": "es6"
    }
}
```

`this` 类型

在方法中返回当前对象 (也就是 `this`) 是一种创建链式 API 的常见方式. 比如, 考虑下面的 `BasicCalculator` 模块:

```
export default class BasicCalculator {
    public constructor(protected value: number = 0) { }

    public currentValue(): number {
        return this.value;
    }

    public add(operand: number) {
        this.value += operand;
        return this;
    }

    public subtract(operand: number) {
        this.value -= operand;
        return this;
    }

    public multiply(operand: number) {
        this.value *= operand;
        return this;
    }

    public divide(operand: number) {
        this.value /= operand;
        return this;
    }
}
```

使用者可以这样表述 `2 * 5 + 1`：

```
import calc from "./BasicCalculator";

let v = new calc(2)
    .multiply(5)
    .add(1)
    .currentValue();
```

这使得这么一种优雅的编码方式成为可能; 然而, 对于想要去继承 `BasicCalculator` 的类来说有一个问题. 想象使用者可能需要编写一个 `ScientificCalculator` :

```
import BasicCalculator from "./BasicCalculator";

export default class ScientificCalculator extends BasicCalculato
r {
    public constructor(value = 0) {
        super(value);
    }

    public square() {
        this.value = this.value ** 2;
        return this;
    }

    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
}
```

因为 `BasicCalculator` 的方法返回了 `this` , TypeScript 过去推断的类型是 `BasicCalculator` , 如果在 `ScientificCalculator` 的实例上调用属于 `BasicCalculator` 的方法, 类型系统不能很好地处理.

举例来说:

```
import calc from "./ScientificCalculator";

let v = new calc(0.5)
    .square()
    .divide(2)
    .sin()    // Error: 'BasicCalculator' 没有 'sin' 方法.
    .currentValue();
```

这已经不再是问题 - TypeScript 现在在类的实例方法中, 会将 `this` 推断为一个特殊的叫做 `this` 的类型. `this` 类型也就写作 `this` , 可以大致理解为 "方法调用时点左边的类型".

`this` 类型在描述一些使用了 mixin 风格继承的库 (比如 Ember.js) 的交叉类型:

```
interface MyType {
    extend<T>(other: T): this & T;
}
```

ES7 幂运算符

TypeScript 1.7 支持将在 ES7/ES2016 中增加的幂运算符: `**` 和 `**=` . 这些运算符会被转换为 ES3/ES5 中的 `Math.pow` .

举例

```
var x = 2 ** 3;
var y = 10;
y **= 2;
var z =  -(4 ** 3);
```

会生成下面的 JavaScript:

```
var x = Math.pow(2, 3);
var y = 10;
y = Math.pow(y, 2);
var z = -(Math.pow(4, 3));
```

改进对象字面量解构的检查

TypeScript 1.7 使对象和数组字面量解构初始值的检查更加直观和自然.

当一个对象字面量通过与之对应的对象解构绑定推断类型时:

- 对象解构绑定中有默认值的属性对于对象字面量来说可选.

- 对象解构绑定中的属性如果在对象字面量中没有匹配的值, 则该属性必须有默认值, 并且会被添加到对象字面量的类型中.
- 对象字面量中的属性必须在对象解构绑定中存在.

当一个数组字面量通过与之对应的数组解构绑定推断类型时:

- 数组解构绑定中的元素如果在数组字面量中没有匹配的值, 则该元素必须有默认值, 并且会被添加到数组字面量的类型中.

举例

```
// f1 的类型为 (arg?: { x?: number, y?: number }) => void
function f1({ x = 0, y = 0 } = {}) { }

// And can be called as:
f1();
f1({});
f1({ x: 1 });
f1({ y: 1 });
f1({ x: 1, y: 1 });

// f2 的类型为 (arg?: (x: number, y?: number) => void
function f2({ x, y = 0 } = { x: 0 }) { }

f2();
f2({});        // 错误, x 非可选
f2({ x: 1 });
f2({ y: 1 });  // 错误, x 非可选
f2({ x: 1, y: 1 });
```

装饰器 (decorators) 支持的编译目标版本增加 ES3

装饰器现在可以编译到 ES3. TypeScript 1.7 在 `__decorate` 函数中移除了 ES5 中增加的 `reduceRight` . 相关改动也内联了对 `Object.getOwnPropertyDescriptor` 和 `Object.defineProperty` 的调用, 并向后兼容, 使 ES5 的输出可以消除前面提到的 `Object` 方法的重复[1].

TypeScript 1.6

JSX 支持

JSX 是一种可嵌入的类似 XML 的语法. 它将最终被转换为合法的 JavaScript, 但转换的语义和具体实现有关. JSX 随着 React 流行起来, 也出现在其他应用中. TypeScript 1.6 支持 JavaScript 文件中 JSX 的嵌入, 类型检查, 以及直接编译为 JavaScript 的选项.

新的 `.tsx` 文件扩展名和 `as` 运算符

TypeScript 1.6 引入了新的 `.tsx` 文件扩展名. 这一扩展名一方面允许 TypeScript 文件中的 JSX 语法, 一方面将 `as` 运算符作为默认的类型转换方式 (避免 JSX 表达式和 TypeScript 前置类型转换运算符之间的歧义). 比如:

```
var x = <any> foo;
// 与如下等价:
var x = foo as any;
```

使用 React

使用 React 及 JSX 支持, 你需要使用 React 类型声明. 这些类型定义了 `JSX` 命名空间, 以便 TypeScript 能正确地检查 React 的 JSX 表达式. 比如:

```
/// <reference path="react.d.ts" />

interface Props {
  name: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent name="bar" />; // 没问题
<MyComponent name={0} />; // 错误, `name` 不是一个字符串
```

使用其他 JSX 框架

JSX 元素的名称和属性是根据 `JSX` 命名空间来检验的. 请查看 JSX 页面了解如何为自己的框架定义 `JSX` 命名空间.

编译输出

TypeScript 支持两种 `JSX` 模式: `preserve` (保留) 和 `react` .

- `preserve` 模式将会在输出中保留 JSX 表达式, 使之后的转换步骤可以处理. 并且输出的文件扩展名为 `.jsx` .
- `react` 模式将会生成 `React.createElement` , 不再需要再通过 JSX 转换即可运行, 输出的文件扩展名为 `.js` .

查看 JSX 页面了解更多 JSX 在 TypeScript 中的使用.

交叉类型 (intersection types)

TypeScript 1.6 引入了交叉类型作为联合类型 (union types) 逻辑上的补充. 联合类型 `A | B` 表示一个类型为 `A` 或 `B` 的实体, 而交叉类型 `A & B` 表示一个类型同时为 `A` 或 `B` 的实体.

例子

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U> {};
    for (let id in first) {
        result[id] = first[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            result[id] = second[id];
        }
    }
    return result;
}

var x = extend({ a: "hello" }, { b: 42 });
var s = x.a;
var n = x.b;
```

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
interface A { a: string }
interface B { b: string }
interface C { c: string }

var abc: A & B & C;
abc.a = "hello";
abc.b = "hello";
abc.c = "hello";
```

查看 issue #1256 了解更多.

本地类型声明

本地的类, 接口, 枚举和类型别名现在可以在函数声明中出现. 本地类型为块级作用域, 与 `let` 和 `const` 声明的变量类似. 比如说:

```
function f() {
    if (true) {
        interface T { x: number }
        let v: T;
        v.x = 5;
    }
    else {
        interface T { x: string }
        let v: T;
        v.x = "hello";
    }
}
```

推导出的函数返回值类型可能在函数内部声明的. 调用函数的地方无法引用到这样的本地类型, 但是它当然能从类型结构上匹配. 比如:

```
interface Point {
    x: number;
    y: number;
}

function getPointFactory(x: number, y: number) {
    class P {
        x = x;
        y = y;
    }
    return P;
}

var PointZero = getPointFactory(0, 0);
var PointOne = getPointFactory(1, 1);
var p1 = new PointZero();
var p2 = new PointZero();
var p3 = new PointOne();
```

本地的类型可以引用类型参数，本地的类和接口本身即可能是泛型。比如:

```
function f3() {
    function f<X, Y>(x: X, y: Y) {
        class C {
            public x = x;
            public y = y;
        }
        return C;
    }
    let C = f(10, "hello");
    let v = new C();
    let x = v.x;  // number
    let y = v.y;  // string
}
```

类表达式

TypeScript 1.6 增加了对 ES6 类表达式的支持. 在一个类表达式中, 类的名称是可选的, 如果指明, 作用域仅限于类表达式本身. 这和函数表达式可选的名称类似. 在类表达式外无法引用其实例类型, 但是自然也能够从类型结构上匹配. 比如:

```
let Point = class {
    constructor(public x: number, public y: number) { }
    public length() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
};
var p = new Point(3, 4);  // p has anonymous class type
console.log(p.length());
```

继承表达式

TypeScript 1.6 增加了对类继承任意值为一个构造函数的表达式的支持. 这样一来内建的类型也可以在类的声明中被继承.

`extends` 语句过去需要指定一个类型引用, 现在接受一个可选类型参数的表达式. 表达式的类型必须为有至少一个构造函数签名的构造函数, 并且需要和 `extends` 语句中类型参数数量一致. 匹配的构造函数签名的返回值类型是类实例类型继承的基类型. 如此一来, 这使得普通的类和与类相似的表达式可以在 `extends` 语句中使用.

一些例子:

```
// 继承内建类

class MyArray extends Array<number> { }
class MyError extends Error { }

// 继承表达式类

class ThingA {
    getGreeting() { return "Hello from A"; }
}

class ThingB {
    getGreeting() { return "Hello from B"; }
}

interface Greeter {
    getGreeting(): string;
}

interface GreeterConstructor {
    new (): Greeter;
}

function getGreeterBase(): GreeterConstructor {
    return Math.random() >= 0.5 ? ThingA : ThingB;
}

class Test extends getGreeterBase() {
    sayHello() {
        console.log(this.getGreeting());
    }
}
```

`abstract` (抽象的) 类和方法

TypeScript 1.6 为类和它们的方法增加了 `abstract` 关键字. 一个抽象类允许没有被实现的方法, 并且不能被构造.

例子

```
abstract class Base {
    abstract getThing(): string;
    getOtherThing() { return 'hello'; }
}

let x = new Base(); // 错误, 'Base' 是抽象的

// 错误, 必须也为抽象类, 或者实现 'getThing' 方法
class Derived1 extends Base { }

class Derived2 extends Base {
    getThing() { return 'hello'; }
    foo() {
        super.getThing();// 错误: 不能调用 'super' 的抽象方法
    }
}

var x = new Derived2(); // 正确
var y: Base = new Derived2(); // 同样正确
y.getThing(); // 正确
y.getOtherThing(); // 正确
```

泛型别名

TypeScript 1.6 中, 类型别名支持泛型. 比如:

```
type Lazy<T> = T | (() => T);

var s: Lazy<string>;
s = "eager";
s = () => "lazy";

interface Tuple<A, B> {
    a: A;
    b: B;
}

type Pair<T> = Tuple<T, T>;
```

更严格的对象字面量赋值检查

为了能发现多余或者错误拼写的属性，TypeScript 1.6 使用了更严格的对象字面量检查. 确切地说，在将一个新的对象字面量赋值给一个变量，或者传递给类型非空的参数时，如果对象字面量的属性在目标类型中不存在，则会视为错误.

例子

```
var x: { foo: number };
x = { foo: 1, baz: 2 };  // 错误, 多余的属性 `baz`

var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 };  // 错误, 多余或者拼错的属性 `baz`
```

一个类型可以通过包含一个索引签名来显示指明未出现在类型中的属性是被允许的.

```
var x: { foo: number, [x: string]: any };
x = { foo: 1, baz: 2 };  // 现在 `baz` 匹配了索引签名
```

ES6 生成器 (generators)

TypeScript 1.6 添加了对于 ES6 输出的生成器支持.

一个生成器函数可以有返回值类型标注, 就像普通的函数. 标注表示生成器函数返回的生成器的类型. 这里有个例子:

```
function *g(): Iterable<string> {
    for (var i = 0; i < 100; i++) {
        yield ""; // string 可以赋值给 string
    }
    yield * otherStringGenerator(); // otherStringGenerator 必须
可遍历, 并且元素类型需要可赋值给 string
}
```

没有标注类型的生成器函数会有自动推演的类型. 在下面的例子中, 类型会由 yield 语句推演出来:

```
function *g() {
    for (var i = 0; i < 100; i++) {
        yield ""; // 推导出 string
    }
    yield * otherStringGenerator(); // 推导出 otherStringGenerato
r 的元素类型
}
```

对 `async` (异步) 函数的试验性支持

TypeScript 1.6 增加了编译到 ES6 时对 `async` 函数试验性的支持. 异步函数会执行一个异步的操作, 在等待的同时不会阻塞程序的正常运行. 这是通过与 ES6 兼容的 `Promise` 实现完成的, 并且会将函数体转换为支持在等待的异步操作完成时继续的形式.

由 `async` 标记的函数或方法被称作异步函数. 这个标记告诉了编译器该函数体需要被转换, 关键字 *await* 则应该被当做一个一元运算符, 而不是标示符. 一个异步函数必须返回类型与 `Promise` 兼容的值. 返回值类型的推断只能在有一个全局的, 与 ES6 兼容的 `Promise` 类型时使用.

例子

```
var p: Promise<number> = /* ... */;
async function fn(): Promise<number> {
  var i = await p; // 暂停执行直到 'p' 得到结果. 'i' 的类型为 "number"

  return 1 + i;
}

var a = async (): Promise<number> => 1 + await p; // 暂停执行.
var a = async () => 1 + await p; // 暂停执行. 使用 --target ES6 选
项编译时返回值类型被推断为 "Promise<number>"
var fe = async function(): Promise<number> {
  var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "number"

  return 1 + i;
}

class C {
  async m(): Promise<number> {
    var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "numb
er"
    return 1 + i;
  }

  async get p(): Promise<number> {
    var i = await p; // 暂停执行知道 'p' 得到结果. 'i' 的类型为 "numb
er"
    return 1 + i;
  }
}
```

每天发布新版本

由于并不算严格意义上的语言变化[2], 每天的新版本可以使用如下命令安装获得:

```
npm install -g typescript@next
```

对模块解析逻辑的调整

从 1.6 开始, TypeScript 编译器对于 "commonjs" 的模块解析会使用一套不同的规则. 这些规则 尝试模仿 Node 查找模块的过程. 这就意味着 node 模块可以包含它的类型信息, 并且 TypeScript 编译器可以找到这些信息. 不过用户可以通过使用 `--moduleResolution` 命令行选项覆盖模块解析规则. 支持的值有:

- 'classic' - TypeScript 1.6 以前的编译器使用的模块解析规则
- 'node' - 与 node 相似的模块解析

合并外围类和接口的声明

外围类的实例类型可以通过接口声明来扩展. 类构造函数对象不会被修改. 比如说:

```
declare class Foo {
    public x : number;
}

interface Foo {
    y : string;
}

function bar(foo : Foo)  {
    foo.x = 1; // 没问题, 在类 Foo 中有声明
    foo.y = "1"; // 没问题, 在接口 Foo 中有声明
}
```

用户定义的类型收窄函数

TypeScript 1.6 增加了一个新的在 `if` 语句中收窄变量类型的方式, 作为对 `typeof` 和 `instanceof` 的补充. 用户定义的类型收窄函数的返回值类型标注形式为 `x is T` , 这里 `x` 是函数声明中的形参, `T` 是任何类型. 当一个用户定义的类型收窄函数在 `if` 语句中被传入某个变量执行时, 该变量的类型会被收窄到 `T` .

例子

```
function isCat(a: any): a is Cat {
  return a.name === 'kitty';
}

var x: Cat | Dog;
if(isCat(x)) {
  x.meow(); // 那么, x 在这个代码块内是 Cat 类型
}
```

`tsconfig.json` 对 `exclude` 属性的支持

一个没有写明 `files` 属性的 `tsconfig.json` 文件 (默认会引用所有子目录下的 *.ts 文件) 现在可以包含一个 `exclude` 属性, 指定需要在编译中排除的文件或者目录列表. `exclude` 属性必须是一个字符串数组, 其中每一个元素指定对应的一个文件或者文件夹名称对于 `tsconfig.json` 文件所在位置的相对路径. 举例来说:

```
{
    "compilerOptions": {
        "out": "test.js"
    },
    "exclude": [
        "node_modules",
        "test.ts",
        "utils/t2.ts"
    ]
}
```

`exclude` 列表不支持通配符. 仅仅可以是文件或者目录的列表.

`--init` 命令行选项

在一个目录中执行 `tsc --init` 可以在该目录中创建一个包含了默认值的 `tsconfig.json` . 可以通过一并传递其他选项来生成初始的 `tsconfig.json` .

TypeScript 1.5

ES6 模块

TypeScript 1.5 支持 ECMAScript 6 (ES6) 模块. ES6 模块可以看做之前 TypeScript 的外部模块换上了新的语法: ES6 模块是分开加载的源文件, 这些文件还可能引入其他模块, 并且导出部分供外部可访问. ES6 模块新增了几种导入和导出声明. 我们建议使用 TypeScript 开发的库和应用能够更新到新的语法, 但不做强制要求. 新的 ES6 模块语法和 TypeScript 原来的内部和外部模块结构同时被支持, 如果需要也可以混合使用.

导出声明

作为 TypeScript 已有的 `export` 前缀支持, 模块成员也可以使用单独导出的声明导出, 如果需要, `as` 语句可以指定不同的导出名称.

```
interface Stream { ... }
function writeToStream(stream: Stream, data: string) { ... }
export { Stream, writeToStream as write };  // writeToStream 导出为 write
```

引入声明也可以使用 `as` 语句来指定一个不同的导入名称. 比如:

```
import { read, write, standardOutput as stdout } from "./inout";
var s = read(stdout);
write(stdout, s);
```

作为单独导入的候选项, 命名空间导入可以导入整个模块:

```
import * as io from "./inout";
var s = io.read(io.standardOutput);
io.write(io.standardOutput, s);
```

重新导出

使用 `from` 语句一个模块可以复制指定模块的导出项到当前模块, 而无需创建本地名称.

```
export { read, write, standardOutput as stdout } from "./inout";
```

`export *` 可以用来重新导出另一个模块的所有导出项. 在创建一个聚合了其他几个模块导出项的模块时很方便.

```
export function transform(s: string): string { ... }
export * from "./mod1";
export * from "./mod2";
```

默认导出项

一个 export default 声明表示一个表达式是这个模块的默认导出项.

```
export default class Greeter {
    sayHello() {
        console.log("Greetings!");
    }
}
```

对应的可以使用默认导入:

```
import Greeter from "./greeter";
var g = new Greeter();
g.sayHello();
```

无导入加载

"无导入加载" 可以被用来加载某些只需要其副作用的模块.

```
import "./polyfills";
```

了解更多关于模块的信息，请参见 ES6 模块支持规范.

声明与赋值的解构

TypeScript 1.5 添加了对 ES6 解构声明与赋值的支持.

解构

解构声明会引入一个或多个命名变量，并且初始化它们的值为对象的属性或者数组的元素对应的值.

比如说，下面的例子声明了变量 `x`，`y` 和 `z`，并且分别将它们的值初始化为 `getSomeObject().x`，`getSomeObject().y` 和 `getSomeObject().z`：

```
var { x, y, z } = getSomeObject();
```

解构声明也可以用于从数组中得到值.

```
var [x, y, z = 10] = getSomeArray();
```

相似的，解构可以用在函数的参数声明中：

```
function drawText({ text = "", location: [x, y] = [0, 0], bold = false }) {
    // 画出文本
}

// 以一个对象字面量为参数调用 drawText
var item = { text: "someText", location: [1,2,3], style: "italics" };
drawText(item);
```

赋值

解构也可以被用于普通的赋值表达式. 举例来讲, 交换两个变量的值可以被写作一个解构赋值:

```
var x = 1;
var y = 2;
[x, y] = [y, x];
```

`namespace` (命名空间) 关键字

过去 TypeScript 中 `module` 关键字既可以定义 "内部模块", 也可以定义 "外部模块"; 这让刚刚接触 TypeScript 的开发者有些困惑. "内部模块" 的概念更接近于大部分人眼中的命名空间; 而 "外部模块" 对于 JS 来讲, 现在也就是模块了.

> 注意: 之前定义内部模块的语法依然被支持.

之前:

```
module Math {
    export function add(x, y) { ... }
}
```

之后:

```
namespace Math {
    export function add(x, y) { ... }
}
```

`let` 和 `const` 的支持

ES6 的 `let` 和 `const` 声明现在支持编译到 ES3 和 ES5.

Const

```
const MAX = 100;

++MAX; // 错误: 自增/减运算符不能用于一个常量
```

块级作用域

```
if (true) {
  let a = 4;
  // 使用变量 a
}
else {
  let a = "string";
  // 使用变量 a
}

alert(a); // 错误: 变量 a 在当前作用域未定义
```

`for...of` 的支持

TypeScript 1.5 增加了 ES6 `for...of` 循环编译到 ES3/ES5 时对数组的支持, 以及编译到 ES6 时对满足 `Iterator` 接口的全面支持.

例子

TypeScript 编译器会转译 `for...of` 数组到具有语义的 ES3/ES5 JavaScript (如果被设置为编译到这些版本).

```
for (var v of expr) { }
```

会输出为:

```
for (var _i = 0, _a = expr; _i < _a.length; _i++) {
    var v = _a[_i];
}
```

装饰器

> TypeScript 装饰器是局域 ES7 装饰器 提案的.

一个装饰器是:

- 一个表达式
- 并且值为一个函数
- 接受 `target` , `name` , 以及属性描述对象作为参数
- 可选返回一个会被应用到目标对象的属性描述对象

> 了解更多, 请参见 装饰器 提案.

例子

装饰器 `readonly` 和 `enumerable(false)` 会在属性 `method` 添加到类 `C` 上之前被应用. 这使得装饰器可以修改其实现, 具体到这个例子, 设置了 `descriptor` 为 `writable: false` 以及 `enumerable: false` .

```
class C {
  @readonly
  @enumerable(false)
  method() { }
}

function readonly(target, key, descriptor) {
    descriptor.writable = false;
}

function enumerable(value) {
  return function (target, key, descriptor) {
     descriptor.enumerable = value;
  }
}
```

计算属性

使用动态的属性初始化一个对象可能会很麻烦. 参考下面的例子:

```
type NeighborMap = { [name: string]: Node };
type Node = { name: string; neighbors: NeighborMap;}

function makeNode(name: string, initialNeighbor: Node): Node {
    var neighbors: NeighborMap = {};
    neighbors[initialNeighbor.name] = initialNeighbor;
    return { name: name, neighbors: neighbors };
}
```

这里我们需要创建一个包含了 neighbor-map 的变量, 便于我们初始化它. 使用 TypeScript 1.5, 我们可以让编译器来干重活:

```
function makeNode(name: string, initialNeighbor: Node): Node {
    return {
        name: name,
        neighbors: {
            [initialNeighbor.name]: initialNeighbor
        }
    }
}
```

指出 `UMD` 和 `System` 模块输出

作为 `AMD` 和 `CommonJS` 模块加载器的补充, TypeScript 现在支持输出为 `UMD` (Universal Module Definition) 和 `System` 模块的格式.

用法:

> tsc --module umd

以及

> tsc --module system

Unicode 字符串码位转义

ES6 中允许用户使用单个转义表示一个 Unicode 码位.

举个例子，考虑我们需要转义一个包含了字符 " 的字符串. 在 UTF-16/USC2 中，" 被表示为一个代理对，意思就是它被编码为一对 16 位值的代码单元，具体来说是 `0xD842` 和 `0xDFB7`. 之前这意味着你必须将该码位转义为 `"\uD842\uDFB7"`. 这样做有一个重要的问题，就事很难讲两个独立的字符同一个代理对区分开来.

通过 ES6 的码位转义，你可以在字符串或模板字符串中清晰地通过一个转义表示一个确切的字符: `"\u{20bb7}"`. TypeScript 在编译到 ES3/ES5 时会将该字符串输出为 `"\uD842\uDFB7"`.

标签模板字符串编译到 ES3/ES5

TypeScript 1.4 中，我们添加了模板字符串编译到所有 ES 版本的支持，并且支持标签模板字符串编译到 ES6. 得益于 @ivogabe 的大量付出，我们填补了标签模板字符串对编译到 ES3/ES5 的支持.

当编译到 ES3/ES5 时，下面的代码:

```
function oddRawStrings(strs: TemplateStringsArray, n1, n2) {
    return strs.raw.filter((raw, index) => index % 2 === 1);
}

oddRawStrings `Hello \n${123} \t ${456}\n world`
```

会被输出为:

```
function oddRawStrings(strs, n1, n2) {
    return strs.raw.filter(function (raw, index) {
        return index % 2 === 1;
    });
}
(_a = ["Hello \n", " \t ", "\n world"], _a.raw = ["Hello \\n", "
 \\t ", "\\n world"], oddRawStrings(_a, 123, 456));
var _a;
```

AMD 可选依赖名称

`/// <amd-dependency path="x" />` 会告诉编译器需要被注入到模块 `require` 方法中的非 TS 模块依赖; 然而在 TS 代码中无法使用这个模块.

新的 `amd-dependency name` 属性允许为 AMD 依赖传递一个可选的名称.

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

生成的 JS 代码:

```
define(["require", "exports", "legacy/moduleA"], function (require, exports, moduleA) {
    moduleA.callStuff()
});
```

通过 `tsconfig.json` 指示一个项目

通过添加 `tsconfig.json` 到一个目录指明这是一个 TypeScript 项目的根目录. `tsconfig.json` 文件指定了根文件以及编译项目需要的编译器选项. 一个项目可以由以下方式编译:

- 调用 tsc 并不指定输入文件, 此时编译器会从当前目录开始往上级目录寻找 `tsconfig.json` 文件.
- 调用 tsc 并不指定输入文件, 使用 `-project` (或者 `-p`) 命令行选项指定包含了 `tsconfig.json` 文件的目录.

例子

```
{
    "compilerOptions": {
        "module": "commonjs",
        "noImplicitAny": true,
        "sourceMap": true,
    }
}
```

参见 tsconfig.json wiki 页面 查看更多信息.

`--rootDir` 命令行选项

选项 `--outDir` 在输出中会保留输入的层级关系. 编译器将所有输入文件共有的最长路径作为根路径; 并且在输出中应用对应的子层级关系.

有的时候这并不是期望的结果, 比如输入 `FolderA\FolderB\1.ts` 和 `FolderA\FolderB\2.ts`, 输出结构会是 `FolderA\FolderB\` 对应的结构. 如果输入中新增 `FolderA\3.ts` 文件, 输出的结构将突然变为 `FolderA\` 对应的结构.

`--rootDir` 指定了会输出对应结构的输入目录, 不再通过计算获得.

`--noEmitHelpers` 命令行选项

TypeScript 编译器在需要的时候会输出一些像 `__extends` 这样的工具函数. 这些函数会在使用它们的所有文件中输出. 如果你想要聚合所有的工具函数到同一个位置, 或者覆盖默认的行为, 使用 `--noEmitHelpers` 来告知编译器不要输出它们.

`--newLine` 命令行选项

默认输出的换行符在 Windows 上是 `\r\n`, 在 *nix 上是 `\n`. `--newLine` 命令行标记可以覆盖这个行为, 并指定输出文件中使用的换行符.

`--inlineSourceMap` and `inlineSources` 命令行选项

`--inlineSourceMap` 将内嵌源文件映射到 `.js` 文件, 而不是在单独的 `.js.map` 文件中. `--inlineSources` 允许进一步将 `.ts` 文件内容包含到输出文件中.

TypeScript 1.4

联合类型

概述

联合类型有助于表示一个值的类型可以是多种类型之一的情况。比如，有一个API接命令行传入 `string` 类型， `string[]` 类型或者是一个返回 `string` 的函数。你就可以这样写：

```
interface RunOptions {
   program: string;
   commandline: string[]|string|(() => string);
}
```

给联合类型赋值也很直观 -- 只要这个值能满足联合类型中任意一个类型那么就可以赋值给这个联合类型：

```
var opts: RunOptions = /* ... */;
opts.commandline = '-hello world'; // OK
opts.commandline = ['-hello', 'world']; // OK
opts.commandline = [42]; // Error, 数字不是字符串或字符串数组
```

当读取联合类型时，你可以访问类型共有的属性：

```
if(opts.length === 0) { // OK, string和string[]都有'length'属性
  console.log("it's empty");
}
```

使用类型保护，你可以轻松地使用联合类型：

```
function formatCommandline(c: string|string[]) {
    if(typeof c === 'string') {
        return c.trim();
    } else {
        return c.join(' ');
    }
}
```

严格的泛型

随着联合类型可以表示有很多类型的场景，我们决定去改进泛型调用的规范性。之前，这段代码编译不会报错（出乎意料）：

```
function equal<T>(lhs: T, rhs: T): boolean {
  return lhs === rhs;
}

// 之前没有错误
// 现在会报错：在string和number之前没有最佳的基本类型
var e = equal(42, 'hello');
```

通过联合类型，你可以指定你想要的行为，在函数定义时或在调用的时候：

```
// 'choose' function where types must match
function choose1<T>(a: T, b: T): T { return Math.random() > 0.5
? a : b }
var a = choose1('hello', 42); // Error
var b = choose1<string|number>('hello', 42); // OK

// 'choose' function where types need not match
function choose2<T, U>(a: T, b: U): T|U { return Math.random() >
0.5 ? a : b }
var c = choose2('bar', 'foo'); // OK, c: string
var d = choose2('hello', 42); // OK, d: string|number
```

更好的类型推断

当一个集合里有多种类型的值时，联合类型会为数组或其它地方提供更好的类型推断：

```
var x = [1, 'hello']; // x: Array<string|number>
x[0] = 'world'; // OK
x[0] = false; // Error, boolean is not string or number
```

`let` 声明

在JavaScript里， `var` 声明会被“提升”到所在作用域的顶端。这可能会引发一些让人不解的bugs：

```
console.log(x); // meant to write 'y' here
/* later in the same block */
var x = 'hello';
```

TypeScript已经支持新的ES6的关键字 `let` ，声明一个块级作用域的变量。一个 `let` 变量只能在声明之后的位置被引用，并且作用域为声明它的块里：

```
if(foo) {
    console.log(x); // Error, cannot refer to x before its declaration
    let x = 'hello';
} else {
    console.log(x); // Error, x is not declared in this block
}
```

`let` 只在设置目标为ECMAScript 6 （ `--target ES6` ）时生效。

`const` 声明

另一个TypeScript支持的ES6里新出现的声明类型是 `const` 。不能给一个 `const` 类型变量赋值，只能在声明的时候初始化。这对于那些在初始化之后就不想去改变它的值的情况下是很有帮助的：

```
const halfPi = Math.PI / 2;
halfPi = 2; // Error, can't assign to a `const`
```

`const` 只在设置目标为ECMAScript 6 （ `--target ES6` ）时生效。

模版字符串

TypeScript现已支持ES6模块字符串。通过它可以方便地在字符串中嵌入任何表达式：

```
var name = "TypeScript";
var greeting  = `Hello, ${name}! Your name has ${name.length} ch
aracters`;
```

当编译目标为ES6之前的版本时，这个字符串被分解为：

```
var name = "TypeScript!";
var greeting = "Hello, " + name + "! Your name has " + name.leng
th + " characters";
```

类型守护

JavaScript常用模式之一是在运行时使用 `typeof` 或 `instanceof` 检查表达式的类型。 在 `if` 语句里使用它们的时候，TypeScript可以识别出这些条件并且随之改变类型推断的结果。

使用 `typeof` 来检查一个变量：

```
var x: any = /* ... */;
if(typeof x === 'string') {
    console.log(x.subtr(1)); // Error, 'subtr' does not exist on
 'string'
}
// x is still any here
x.unknown(); // OK
```

结合联合类型使用 `typeof` 和 `else` ：

```
var x: string|HTMLElement = /* ... */;
if(typeof x === 'string') {
    // x is string here, as shown above
} else {
    // x is HTMLElement here
    console.log(x.innerHTML);
}
```

结合类和联合类型使用 `instanceof` ：

```
class Dog { woof() { } }
class Cat { meow() { } }
var pet: Dog|Cat = /* ... */;
if(pet instanceof Dog) {
    pet.woof(); // OK
} else {
    pet.woof(); // Error
}
```

类型别名

你现在可以使用 `type` 关键字来为类型定义一个“别名”：

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

类型别名与其原始的类型完全一致；它们只是简单的替代名。

`const enum`（完全嵌入的枚举）

枚举很有帮助，但是有些程序实际上并不需要它生成的代码并且想要将枚举变量所代码的数字值直接替换到对应位置上。新的 `const enum` 声明与正常的 `enum` 在类型安全方面具有同样的作用，只是在编译时会清除掉。

```
const enum Suit { Clubs, Diamonds, Hearts, Spades }
var d = Suit.Diamonds;
```

Compiles to exactly:

```
var d = 1;
```

TypeScript也会在可能的情况下计算枚举值：

```
enum MyFlags {
  None = 0,
  Neat = 1,
  Cool = 2,
  Awesome = 4,
  Best = Neat | Cool | Awesome
}
var b = MyFlags.Best; // emits var b = 7;
```

`-noEmitOnError` 命令行选项

TypeScript编译器的默认行为是当存在类型错误（比如，将 `string` 类型赋值给 `number` 类型）时仍会生成.js文件。这在构建服务器上或是其它场景里可能会是不想看到的情况，因为希望得到的是一次“纯净”的构建。新的 `noEmitOnError` 标记可以阻止在编译时遇到错误的情况下继续生成.js代码。

它现在是MSBuild工程的默认行为；这允许MSBuild持续构建以我们想要的行为进行，输出永远是来自纯净的构建。

AMD 模块名

默认情况下AMD模块以匿名形式生成。这在使用其它工具（比如，r.js）处理生成的模块的时可能会带来麻烦。

新的 `amd-module name` 标签允许给编译器传入一个可选的模块名：

```
//// [amdModule.ts]
///<amd-module name='NamedModule'/>
export class C {
}
```

结果会把 `NamedModule` 赋值成模块名，做为调用AMD `define` 的一部分：

```
//// [amdModule.js]
define("NamedModule", ["require", "exports"], function (require, exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

TypeScript 1.3

受保护的

类里面新的 `protected` 修饰符作用与其它语言如C++，C#和Java中的一样。一个类的 `protected` 成员只在这个类的子类中可见：

```
class Thing {
  protected doSomething() { /* ... */ }
}

class MyThing extends Thing {
  public myMethod() {
    // OK，可以在子类里访问受保护的成员
    this.doSomething();
  }
}
var t = new MyThing();
t.doSomething(); // Error，不能在类外部访问受保护成员
```

元组类型

元组类型表示一个数组，其中元素的类型都是已知的，但是不一样是同样的类型。比如，你可能想要表示一个第一个元素是 `string` 类型第二个元素是 `number` 类型的数组：

```
// Declare a tuple type
var x: [string, number];
// 初始化
x = ['hello', 10]; // OK
// 错误的初始化
x = [10, 'hello']; // Error
```

但是访问一个已知的索引，会得到正确的类型：

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number'没有'substr'方法
```

注意在TypeScript1.4里，当访问超出已知索引的元素时，会返回联合类型：

```
x[3] = 'world'; // OK
console.log(x[5].toString()); // OK, 'string'和'number'都有toStri
ng
x[6] = true; // Error, boolean不是number或string
```

TypeScript 1.1

改进性能

1.1版本的编译器速度比所有之前发布的版本快4倍。阅读这篇博客里的有关图表

更好的模块可见性规则

TypeScript现在只在使用 `--declaration` 标记时才严格强制模块里类型的可见性。这在Angular里很有用，例如：

```
module MyControllers {
  interface ZooScope extends ng.IScope {
    animals: Animal[];
  }
  export class ZooController {
    // Used to be an error (cannot expose ZooScope), but now is
only
    // an error when trying to generate .d.ts files
    constructor(public $scope: ZooScope) { }
    /* more code */
  }
}
```

Breaking Changes

- TypeScript 2.3
- TypeScript 2.2
- TypeScript 2.1
- TypeScript 2.0
- TypeScript 1.8
- TypeScript 1.7
- TypeScript 1.6
- TypeScript 1.5
- TypeScript 1.4
- TypeScript 1.1

TypeScript 2.3

完整的破坏性改动列表请到这里查看:breaking change issues.

空的泛型列表会被标记为错误

示例

```
class X<> {}  // Error: Type parameter list cannot be empty.
function f<>() {}  // Error: Type parameter list cannot be empty.

const x: X<> = new X<>();  // Error: Type parameter list cannot be empty.
```

TypeScript 2.2

完整的破坏性改动列表请到这里查看:breaking change issues.

标准库里的DOM API变动

- 现在标准库里有 `Window.fetch` 的声明；仍依赖于 `@types\whatwg-fetch` 会产生声明冲突错误，需要被移除。
- 现在标准库里有 `ServiceWorker` 的声明；仍依赖于 `@types\service_worker_api` 会产生声明冲突错误，需要被移除。

TypeScript 2.1

完整的破坏性改动列表请到这里查看:breaking change issues.

生成的构造函数代码将 `this` 的值替换为 `super(...)` 调用的返回值

在ES2015中，如果构造函数返回一个对象，那么对于任何 `super(...)` 的调用者将隐式地替换掉 `this` 的值。 因此，有必要获取任何可能的 `super(...)` 的返回值并用 `this` 进行替换。

示例

定义一个类 `C` ：

```
class C extends B {
    public a: number;
    constructor() {
        super();
        this.a = 0;
    }
}
```

将生成如下代码：

```
var C = (function (_super) {
    __extends(C, _super);
    function C() {
        var _this = _super.call(this) || this;
        _this.a = 0;
        return _this;
    }
    return C;
}(B));
```

注意：

- `_super.call(this)` 存入局部变量 `_this`
- 构造函数体里所有使用 `this` 的地方都被替换为 `super` 调用的返回值（例如 `_this` ）
- 每个构造函数将明确地返回它的 `this` ，以确保正确的继承

值得注意的是在 `super(...)` 调用前就使用 `this` 从TypeScript 1.8开始将会引发错误。

继承内置类型如 `Error` ， `Array` 和 `Map` 将是无效的

做为将 `this` 的值替换为 `super(...)` 调用返回值的一部分，子类化 `Error` ， `Array` 等的结果可以是非预料的。 这是因为 `Error` ， `Array` 等的构造函数会使用ECMAScript 6的 `new.target` 来调整它们的原型链； 然而，在ECMAScript 5中调用构造函数时却没有有效的方法来确保 `new.target` 的值。 在默认情况下，其它低级别的编译器也普遍存在这个限制。

示例

针对如下的子类：

```
class FooError extends Error {
    constructor(m: string) {
        super(m);
    }
    sayHello() {
        return "hello " + this.message;
    }
}
```

你会发现：

- 由这个子类构造出来的对象上的方法可能为 `undefined` ，因此调用 `sayHello` 会引发错误。
- `instanceof` 应用于子类与其实例之前会失效，因此 `(new FooError()) instanceof FooError` 会返回 `false` 。

推荐

做为一个推荐，你可以在任何 `super(...)` 调用后立即手动地调整原型。

```
class FooError extends Error {
    constructor(m: string) {
        super(m);

        // Set the prototype explicitly.
        Object.setPrototypeOf(this, FooError.prototype);
    }

    sayHello() {
        return "hello " + this.message;
    }
}
```

但是，任何 `FooError` 的子类也必须要手动地设置原型。 对于那些不支持 `Object.setPrototypeOf` 的运行时环境，你可以使用 `__proto__` 。

不幸的是，这些变通方法在IE10及其之前的版本.aspx) 你可以手动地将方法从原型上拷贝到实例上（比如从 `FooError.prototype` 到 `this` ），但是原型链却是无法修复的。

`const` 变量和 `readonly` 属性会默认地推断成字面类型

默认情况下， `const` 声明和 `readonly` 属性不会被推断成字符串，数字，布尔和枚举字面量类型。这意味着你的变量/属性可能具有比之前更细的类型。这将体现在使用 `===` 和 `!==` 的时候。

示例

```
const DEBUG = true; // 现在为`true`类型，之前为`boolean`类型

if (DEBUG === false) { // 错误： 操作符'==='不能应用于'true'和'false'

    ...
}
```

推荐

针对故意要求更加宽泛类型的情况下，将类型转换成基础类型：

```
const DEBUG = <boolean>true; // `boolean`类型
```

不对函数和类表达式里捕获的变量进行类型细化

当泛型类型参数具有 `string` ， `number` 或 `boolean` 约束时，会被推断为字符串，数字和布尔字面量类型。此外，如果字面量类型有相同的基础类型（如 `string` ），当没有字面量类型做为推断的最佳超类型时这个规则会失效。

示例

```
declare function push<T extends string>(...args: T[]): T;

var x = push("A", "B", "C"); // 推断成 "A" | "B" | "C" 在TS 2.1, 在TS 2.0里为 string
```

推荐

在调用处明确指定参数类型：

```
var x = push<string>("A", "B", "C"); // x是string
```

没有注解的callback参数如果没有与之匹配的重载参数会触发implicit-any错误

在之前编译器默默地赋予callback（下面的 `c` ）的参数一个 `any` 类型。原因关乎到编译器如何解析重载的函数表达式。从TypeScript 2.1开始，在使用 `--noImplicitAny` 时，这会触发一个错误。

示例

```
declare function func(callback: () => void): any;
declare function func(callback: (arg: number) => void): any;

func(c => { });
```

推荐

删除第一个重载，因为它实在没什么意义；上面的函数可以使用1个或0个必须参数调用，因为函数可以安全地忽略额外的参数。

```
declare function func(callback: (arg: number) => void): any;

func(c => { });
func(() => { });
```

或者，你可以给callback的参数指定一个明确的类型：

```
func((c:number) => { });
```

逗号操作符使用在无副作用的表达式里时会被标记成错误

大多数情况下，这种在之前是有效的逗号表达式现在是错误。

示例

```
let x = Math.pow((3, 5)); // x = NaN, was meant to be `Math.pow(3, 5)`

// This code does not do what it appears to!
let arr = [];
switch(arr.length) {
  case 0, 1:
    return 'zero or one';
  default:
    return 'more than one';
}
```

推荐

`--allowUnreachableCode` 会禁用产生警告在整个编译过程中。或者，你可以使用 `void` 操作符来镇压这个逗号表达式错误：

```
let a = 0;
let y = (void a, 1); // no warning for `a`
```

标准库里的DOM API变动

- **Node.firstChild**，**Node.lastChild**，**Node.nextSibling**，**Node.previousSibling**，**Node.parentElement**和**Node.parentNode**现在是 `Node | null` 而非 `Node` 。

  查看#11113了解详细信息。

  推荐明确检查 `null` 或使用 `!` 断言操作符（比如 `node.lastChild!` ）。

TypeScript 2.0

完整的破坏性改动列表请到这里查看:breaking change issues.

对函数或类表达式的捕获变量不进行类型细化(narrowing)

类型细化不会在函数，类和lambda表达式上进行。

例子

```
var x: number | string;

if (typeof x === "number") {
    function inner(): number {
        return x; // Error, type of x is not narrowed, c is numb
er | string
    }
    var y: number = x; // OK, x is number
}
```

编译器不知道回调函数什么时候被执行。考虑下面的情况：

```
var x: number | string = "a";
if (typeof x === "string") {
    setTimeout(() => console.log(x.charAt(0)), 0);
}
x = 5;
```

当 `x.charAt()` 被调用的时候把 `x` 的类型当作 `string` 是错误的，事实上它确实不是 `string` 类型。

推荐

使用常量代替：

```
const x: number | string = "a";
if (typeof x === "string") {
    setTimeout(() => console.log(x.charAt(0)), 0);
}
```

泛型参数会进行类型细化

例子

```
function g<T>(obj: T) {
    var t: T;
    if (obj instanceof RegExp) {
         t = obj; // RegExp is not assignable to T
    }
}
```

推荐 可以把局部变量声明为特定类型而不是泛型参数或者使用类型断言。

只有get而没有set的存取器会被自动推断为 `readonly` 属性

例子

```
class C {
  get x() { return 0; }
}

var c = new C();
c.x = 1; // Error Left-hand side is a readonly property
```

推荐

定义一个不对属性写值的setter。

在严格模式下函数声明不允许出现在块(block)里

在严格模式下这已经是一个运行时错误。从TypeScript 2.0开始，它会被标记为编译时错误。

例子

```
if( true ) {
    function foo() {}
}

export = foo;
```

推荐

使用函数表达式代替：

```
if( true ) {
    const foo = function() {}
}
```

`TemplateStringsArray` 现是是不可变的

ES2015模版字符串总是将它们的标签以不可变的类数组对象进行传递，这个对象带有一个 `raw` 属性（同样是不可变的）。TypeScript把这个对象命名为 `TemplateStringsArray` 。

便利的是， `TemplateStringsArray` 可以赋值给 `Array<string>` ，因此你可以利用这个较短的类型来使用标签参数：

```
function myTemplateTag(strs: string[]) {
    // ...
}
```

然而，在TypeScript 2.0，支持用 `readonly` 修饰符表示这些对象是不可变的。这样的话， `TemplateStringsArray` 就变成了不可变的，并且不再可以赋值给 `string[]` 。

推荐

直接使用 `TemplateStringsArray` （或者使用 `ReadonlyArray<string>` ）。

TypeScript 1.8

完整的破坏性改动列表请到这里查看:breaking change issues。

现在生成模块代码时会带有 `"use strict";` 头

在ES6模式下模块总是在严格模式下解析，对于生成目标为非ES6的却不是这样。从TypeScript 1.8开始，生成的模块将总为严格模式。这应该不会对现有的大部分代码产生影响，因为TypeScript把大多数因为严格模式而产生的错误当做编译时错误，但还是有一些在运行时才发生错误的TypeScript代码，比如赋值给 `NaN` ，现在将会直接报错。你可以参考MDN Article学习关于严格模式与非严格模式的区别。

若想禁用这个行为，在命令行里传 `--noImplicitUseStrict` 选项或在tsconfig.json文件里指定。

从模块里导出非局部名称

依据ES6/ES2015规范，从模块里导出非局部名称将会报错。

例子

```
export { Promise }; // Error
```

推荐

在导出之前，使用局部变量声明捕获那个全局名称。

```
const localPromise = Promise;
export { localPromise as Promise };
```

默认启用代码可达性（Reachability）检查

TypeScript 1.8里，我们添加了一些可达性检查来阻止一些种类的错误。特别是：

1. 检查代码的可达性（默认启用，可以通过 `allowUnreachableCode` 编译器选项禁用）

```
function test1() {
    return 1;
    return 2; // error here
}

function test2(x) {
    if (x) {
        return 1;
    }
    else {
        throw new Error("NYI")
    }
    var y = 1; // error here
}
```

2. 检查标签是否被使用（默认启用，可以通过 `allowUnusedLabels` 编译器选项禁用）

```
l: // error will be reported - label `l` is unused
while (true) {
}

(x) => { x:x } // error will be reported - label `x` is unus
ed
```

3. 检查是否函数里所有带有返回值类型注解的代码路径都返回了值（默认启用，可以通过 `noImplicitReturns` 编译器选项禁用）

```
// error will be reported since function does not return any
thing explicitly when `x` is falsy.
function test(x): number {
   if (x) return 10;
}
```

4. 检查控制流是否能进到switch语句的case里（默认禁用，可以通过 `noFallthroughCasesInSwitch` 编译器选项启用）。注意没有语句的case不会被检查。

```
switch(x) {
   // OK
   case 1:
   case 2:
       return 1;
}
switch(x) {
   case 1:
       if (y) return 1;
   case 2:
       return 2;
}
```

如果你看到了这些错误，但是你认为这时的代码是合理的话，你可以通过编译选项来阻止报错。

`--module` 不允许与 `--outFile` 一起出现，除非 `--module` 被指定为 `amd` 或 `system`

之前使用模块指定这两个的时候，会生成空的 `out` 文件且不会报错。

标准库里的DOM API变动

- **ImageData.data**现在的类型为 `Uint8ClampedArray` 而不是 `number[]` 。查看#949。
- **HTMLSelectElement .options**现在的类型为 `HTMLCollection` 而不是 `HTMLSelectElement` 。查看#1558。
- **HTMLTableElement.createCaption**，**HTMLTableElement.createTBody**，**HTMLTableElement.createTFoot**，**HTMLTableElement.createTHead**，**HTMLTableElement.insertRow**，**HTMLTableSectionElement.insertRow**和**HTMLTableElement.insertRow**现在返回 `HTMLTableRowElement` 而不是 `HTMLElement` 。查看#3583。
- **HTMLTableRowElement.insertCell**现在返回 `HTMLTableCellElement` 而不是 `HTMLElement` 查看#3583。
- **IDBObjectStore.createIndex**和**IDBDatabase.createIndex**第二个参数类型为 `IDBObjectStoreParameters` 而不是 `any` 。查看#5932。
- **DataTransferItemList.Item**返回值类型变为 `DataTransferItem` 而不

是 `File` 。查看#6106。

- **Window.open**返回值类型变为 `Window` 而不是 `any` 。查看#6418。
- **WeakMap.clear**被移除。查看#6500。

在super-call之前不允许使用 `this`

ES6不允许在构造函数声明里访问 `this` 。

比如：

```
class B {
    constructor(that?: any) {}
}

class C extends B {
    constructor() {
        super(this);  // error;
    }
}

class D extends B {
    private _prop1: number;
    constructor() {
        this._prop1 = 10;  // error
        super();
    }
}
```

TypeScript 1.7

完整的破坏性改动列表请到这里查看:breaking change issues。

从 `this` 中推断类型发生了变化

在类里，`this` 值的类型将被推断成 `this` 类型。 这意味着随后使用原始类型赋值时可能会发生错误。

例子：

```
class Fighter {
    /** @returns the winner of the fight. */
    fight(opponent: Fighter) {
        let theVeryBest = this;
        if (Math.rand() < 0.5) {
            theVeryBest = opponent; // error
        }
        return theVeryBest
    }
}
```

推荐：

添加类型注解：

```
class Fighter {
    /** @returns the winner of the fight. */
    fight(opponent: Fighter) {
        let theVeryBest: Fighter = this;
        if (Math.rand() < 0.5) {
            theVeryBest = opponent; // no error
        }
        return theVeryBest
    }
}
```

类成员修饰符后面会自动插入分号

关键字 `abstract`，`public`，`protected` 和 `private` 是ECMAScript 3里的保留关键字并适用于自动插入分号机制。之前，在这些关键字出现的行尾，TypeScript是不会插入分号的。现在，这已经被改正了，在上例中 `abstract class D` 不再能够正确地继承 `C` 了，而是声明了一个 `m` 方法和一个额外的属性 `abstract` 。

注意，`async` 和 `declare` 已经能够正确自动插入分号了。

例子：

```
abstract class C {
    abstract m(): number;
}
abstract class D extends C {
    abstract
    m(): number;
}
```

推荐：

在定义类成员时删除关键字后面的换行。通常来讲，要避免依赖于自动插入分号机制。

TypeScript 1.6

完整的破坏性改动列表请到这里查看:breaking change issues。

严格的对象字面量赋值检查

当在给变量赋值或给非空类型的参数赋值时，如果对象字面量里指定的某属性不存在于目标类型中时会得到一个错误。

你可以通过使用--suppressExcessPropertyErrors编译器选项来禁用这个新的严格检查。

例子：

```
var x: { foo: number };
x = { foo: 1, baz: 2 };  // Error, excess property `baz`

var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 };  // Error, excess or misspelled property
 `baz`
```

推荐：

为了避免此错误，不同情况下有不同的补救方法：

如果目标类型接收额外的属性，可以增加一个索引：

```
var x: { foo: number, [x: string]: any };
x = { foo: 1, baz: 2 };  // OK, `baz` matched by index signature
```

如果原始类型是一组相关联的类型，使用联合类型明确指定它们的类型而不是仅指定一个基本类型。

```
let animalList: (Dog | Cat | Turkey)[] = [    // use union type
instead of Animal
    {name: "Milo", meow: true },
    {name: "Pepper", bark: true},
    {name: "koko", gobble: true}
];
```

还有可以明确地转换到目标类型以避免此错误：

```
interface Foo {
    foo: number;
}
interface FooBar {
    foo: number;
    bar: number;
}
var y: Foo;
y = <FooBar>{ foo: 1, bar: 2 };
```

CommonJS的模块解析不再假设路径为相对的

之前，对于 `one.ts` 和 `two.ts` 文件，如果它们在相同目录里，那么在 `two.ts` 里面导入 `"one"` 时是相对于 `one.ts` 的路径的。

TypeScript 1.6在编译CommonJS时， `"one"` 不再等同于"./one"。取而代之的是会相对于合适的 `node_modules` 文件夹进行查找，与Node.js在运行时解析模块相似。更多详情，阅读the issue that describes the resolution algorithm。

例子：

`./one.ts`

```
export function f() {
    return 10;
}
```

`./two.ts`

```
import { f as g } from "one";
```

推荐：

修改所有计划之外的非相对的导入。

`./one.ts`

```
export function f() {
    return 10;
}
```

`./two.ts`

```
import { f as g } from "./one";
```

将 `--moduleResolution` 编译器选项设置为 `classic`。

函数和类声明为默认导出时不再能够与在意义上有交叉的同名实体进行合并

在同一空间内默认导出声明的名字与空间内一实体名相同时会得到一个错误；比如，

```
export default function foo() {
}

namespace foo {
    var x = 100;
}
```

和

```
export default class Foo {
    a: number;
}

interface Foo {
    b: string;
}
```

两者都会报错。

然而，在下面的例子里合并是被允许的，因为命名空间并不具备做为值的意义：

```
export default class Foo {
}

namespace Foo {
}
```

推荐：

为默认导出声明本地变量并使用单独的 `export default` 语句：

```
class Foo {
    a: number;
}

interface foo {
    b: string;
}

export default Foo;
```

更多详情，请阅读the originating issue。

模块体以严格模式解析

按照ES6规范，模块体现在以严格模式进行解析。行为将相当于在模块作用域顶端定义了 `"use strict"` ；它包括限制了把 `arguments` 和 `eval` 做为变量名或参数名的使用，把未来保留字做为变量或参数使用，八进制数字字面量的使用等。

标准库里DOM API的改动

- **MessageEvent**和**ProgressEvent**构造函数希望传入参数；查看issue #4295。
- **ImageData**构造函数希望传入参数；查看issue #4220。
- **File**构造函数希望传入参数；查看issue #3999。

系统模块输出使用批量导出

编译器以系统模块的格式使用新的 `_export` 函数批量导出的变体，它接收任何包含键值对的对象做为参数而不是key, value。

模块加载器需要升级到v0.17.1或更高。

npm包的.js内容从'bin'移到了'lib'

TypeScript的npm包入口位置从 `bin` 移动到了 `lib` ，以防‘node_modules/typescript/bin/typescript.js’通过IIS访问的时候造成阻塞（ `bin` 默认是隐藏段因此IIS会阻止访问这个文件夹）。

TypeScript的npm包不会默认全局安装

TypeScript 1.6从package.json里移除了 `preferGlobal` 标记。如果你依赖于这种行为，请使用 `npm install -g typescript` 。

装饰器做为调用表达式进行检查

从1.6开始，装饰器类型检查更准确了；编译器会将装饰器表达式做为以被装饰的实体做为参数的调用表达式来进行检查。这可能会造成以前的代码报错。

TypeScript 1.5

完整的破坏性改动列表请到这里查看:breaking change issues。

不允许在箭头函数里引用 `arguments`

这是为了遵循ES6箭头函数的语义。之前箭头函数里的 `arguments` 会绑定到箭头函数的参数。参照ES6规范草稿 9.2.12，箭头函数不存在 `arguments` 对象。 从TypeScript 1.5开始，在箭头函数里使用 `arguments` 会被标记成错误以确保你的代码转成ES6时没语义上的错误。

例子：

```
function f() {
    return () => arguments; // Error: The 'arguments' object can not be referenced in an arrow function.
}
```

推荐：

```
// 1. 使用带名字的剩余参数
function f() {
    return (...args) => { args; }
}

// 2. 使用函数表达式
function f() {
    return function(){ arguments; }
}
```

内联枚举引用的改动

对于正常的枚举，在1.5之前，编译器仅会内联常量成员，且成员仅在使用字面量初始化时才被当做是常量。这在判断检举值是使用字面量初始化还是表达式时会行为不一致。从TypeScript 1.5开始，所有非const枚举成员都不会被内联。

例子：

```
var x = E.a;  // previously inlined as "var x = 1; /*E.a*/"

enum E {
   a = 1
}
```

推荐：在枚举声明里添加 `const` 修饰符来确保它总是被内联。更多信息，查看#2183。

上下文的类型将作用于 `super` 和括号表达式

在1.5之前，上下文的类型不会作用于括号表达式内部。这就要求做显示的类型转换，尤其是在必须使用括号来进行表达式转换的场合。

在下面的例子里，`m` 具有上下文的类型，它在之前的版本里是没有的。

```
var x: SomeType = (n) => ((m) => q);
var y: SomeType = t ? (m => m.length) : undefined;

class C extends CBase<string> {
    constructor() {
        super({
            method(m) { return m.length; }
        });
    }
}
```

更多信息，查看#1425和#920。

DOM接口的改动

TypeScript 1.5改进了 `lib.d.ts` 库里的DOM类型。这是自TypeScript 1.0以来第一次大的改动；为了拥抱标准DOM规范，很多特定于IE的定义被移除了，同时添加了新的类型如Web Audio和触摸事件。

变通方案：

你可以使用旧的 `lib.d.ts` 配合新版本的编译器。你需要在你的工程里引入之前版本的一个拷贝。这里是本次改动之前的lib.d.ts文件(TypeScript 1.5-alpha)。

变动列表：

- 属性 `selection` 从 `Document` 类型上移除
- 属性 `clipboardData` 从 `Window` 类型上移除
- 删除接口 `MSEventAttachmentTarget`
- 属性 `onresize` ，`disabled` ，`uniqueID` ，`removeNode` ，`fireEvent` ，`currentStyle` ，`runtimeStyle` 从 `HTMLElement` 类型上移除
- 属性 `url` 从 `Event` 类型上移除
- 属性 `execScript` ，`navigate` ，`item` 从 `Window` 类型上移除
- 属性 `documentMode` ，`parentWindow` ，`createEventObject` 从 `Document` 类型上移除
- 属性 `parentWindow` 从 `HTMLDocument` 类型上移除
- 属性 `setCapture` 被完全移除
- 属性 `releaseCapture` 被完全移除
- 属性 `setAttribute` ，`styleFloat` ，`pixelLeft` 从 `CSSStyleDeclaration` 类型上移除
- 属性 `selectorText` 从 `CSSRule` 类型上移除
- `CSSStyleSheet.rules` 现在是 `CSSRuleList` 类型，而非 `MSCSSRuleList`
- `documentElement` 现在是 `Element` 类型，而非 `HTMLElement`
- `Event` 具有一个新的必需属性 `returnValue`
- `Node` 具有一个新的必需属性 `baseURI`
- `Element` 具有一个新的必需属性 `classList`
- `Location` 具有一个新的必需属性 `origin`
- 属性 `MSPOINTER_TYPE_MOUSE` ，`MSPOINTER_TYPE_TOUCH` 从 `MSPointerEvent` 类型上移除
- `CSSStyleRule` 具有一个新的必需属性 `readonly`
- 属性 `execUnsafeLocalFunction` 从 `MSApp` 类型上移除
- 全局方法 `toStaticHTML` 被移除
- `HTMLCanvasElement.getContext` 现在返回 `CanvasRenderingContext2D | WebGLRenderingContex`

- 移除扩展类型 `Dataview` ， `Weakmap` ， `Map` ， `Set`
- `XMLHttpRequest.send` 具有两个重载 `send(data?: Document): void;` 和 `send(data?: String): void;`
- `window.orientation` 现在是 `string` 类型，而非 `number`
- 特定于IE的 `attachEvent` 和 `detachEvent` 从 `Window` 上移除

以下是被新加的**DOM**类型所部分或全部取代的代码库的代表：

- `DefinitelyTyped/auth0/auth0.d.ts`
- `DefinitelyTyped/gamepad/gamepad.d.ts`
- `DefinitelyTyped/interactjs/interact.d.ts`
- `DefinitelyTyped/webaudioapi/waa.d.ts`
- `DefinitelyTyped/webcrypto/WebCrypto.d.ts`

更多信息，查看完整改动。

类代码体将以严格格式解析

按照ES6规范，类代码体现在以严格模式进行解析。行为将相当于在类作用域顶端定义了 `"use strict"` ；它包括限制了把 `arguments` 和 `eval` 做为变量名或参数名的使用，把未来保留字做为变量或参数使用，八进制数字字面量的使用等。

TypeScript 1.4

完整的破坏性改动列表请到这里查看:breaking change issues。

阅读issue #868以了解更多关于联合类型的破坏性改动。

多个最佳通用类型候选

当有多个最佳通用类型可用时，现在编译器会做出选择（依据编译器的具体实现）而不是直接使用第一个。

```
var a: { x: number; y?: number };
var b: { x: number; z?: number };

// 之前 { x: number; z?: number; }[]
// 现在 { x: number; y?: number; }[]
var bs = [b, a];
```

这会在多种情况下发生。具有一组共享的必需属性和一组其它互斥的（可选或其它）属性，空类型，兼容的签名类型（包括泛型和非泛型签名，当类型参数上应用了 `any` 时）。

推荐 使用类型注解指定你要使用的类型。

```
var bs: { x: number; y?: number; z?: number }[] = [b, a];
```

泛型接口

当在多个T类型的参数上使用了不同的类型时会得到一个错误，就算是添加约束也不行：

```
declare function foo<T>(x: T, y:T): T;
var r = foo(1, ""); // r used to be {}, now this is an error
```

添加约束：

```
interface Animal { x }
interface Giraffe extends Animal { y }
interface Elephant extends Animal { z }
function f<T extends Animal>(x: T, y: T): T { return undefined;
}
var g: Giraffe;
var e: Elephant;
f(g, e);
```

在这里查看详细解释。

推荐 如果这种不匹配的行为是故意为之，那么明确指定类型参数：

```
var r = foo<{}>(1, ""); // Emulates 1.0 behavior
var r = foo<string|number>(1, ""); // Most useful
var r = foo<any>(1, ""); // Easiest
f<Animal>(g, e);
```

或重写函数定义指明就算不匹配也没问题：

```
declare function foo<T,U>(x: T, y:U): T|U;
function f<T extends Animal, U extends Animal>(x: T, y: U): T|U
{ return undefined; }
```

泛型剩余参数

不能再使用混杂的参数类型：

```
function makeArray<T>(...items: T[]): T[] { return items; }
var r = makeArray(1, ""); // used to return {}[], now an error
```

`new Array(...)` 也一样

推荐 声明向后兼容的签名，如果1.0的行为是你想要的：

```
function makeArray<T>(...items: T[]): T[];
function makeArray(...items: {}[]): {}[];
function makeArray<T>(...items: T[]): T[] { return items; }
```

带类型参数接口的重载解析

```
var f10: <T>(x: T, b: () => (a: T) => void, y: T) => T;
var r9 = f10('', () => (a => a.foo), 1); // r9 was any, now this
 is an error
```

推荐 手动指定一个类型参数

```
var r9 = f10<any>('', () => (a => a.foo), 1);
```

类声明与类型表达式以严格模式解析

ECMAScript 2015语言规范(ECMA-262 6th Edition)指明*ClassDeclaration*和*ClassExpression*使用严格模式。 因此，在解析类声明或类表达式时将使用额外的限制。

例如：

```
class implements {}  // Invalid: implements is a reserved word i
n strict mode
class C {
    foo(arguments: any) {   // Invalid: "arguments" is not allow
 as a function argument
        var eval = 10;      // Invalid: "eval" is not allowed as
 the left-hand-side expression
        arguments = [];     // Invalid: arguments object is immu
table
    }
}
```

关于严格模式限制的完整列表，请阅读 Annex C - The Strict Mode of ECMAScript of ECMA-262 6th Edition。